VISIT…

VISIT…

Things To Pay Attention To

This is definitely a rough draft of the book.

We believe we have most of a complete book, once we get the contributions we're looking for from others (see below).

The big question for us is the organization.

We are trying to balance telling people how to apply it based on experience in a way that is helpful rather than just a "collection of stories… you figure what to do with them".

We suspect we may have slightly too much "our version of XP Explained" but we're not sure. The intent is to tell people what the pioneers have experienced, but we need to put all of that in context. We don't think we can assume they've read XP Explained (or any of the other books), but we don't want to rehash and we don't have a problem referring them to the other books.

We have stories in all sorts of forms intermingled in here. We should probably stick to no more than 2-3 forms (short blurbs, longer side bars, ???).

Where should the stories go? How do they affect the flow positively or negatively?

Keeping in mind that we want this book to ooze experience, not theory or predictions or unsubstantiated opinions, we want your help in at least one of two roles:

Reviewers

Is this differentiated enough from other XP books to provide clear value?

Does it achieve the goal of experience over theory?

Tell us what you think about the current "macro" flow and "micro" flow (i.e. we'll take either, but please distinguish between book flow, section flow, chapter flow comments)

Is there anything missing?

Content Contributors

The following table summarizes our thoughts on what we'd like individuals to provide based on what we know of your experiences. We'd like to keep each of these

contributions as short as possible, but long enough to pass the test of "communicating experience in a way those that come after you can learn from". Please do not rehash things that are covered elsewhere in the book unless leaving it out would cause the test to fail.

Keep in mind the contributions we expect to receive from others on the list.

If you feel there is something else that you can contribute that is unique, we invite you to contact us and discuss the possibility of its inclusion.

NOTE: We need to get your contributions no later than Friday April 20, 2001

Desired Content Contributions

Here are our initial thoughts about who might contribute some more content. It is not expected to be complete, nor do we assume that everyone on this list will actually contribute:

Contrib itor	Topic	Location (most likely ?)
Travis Griggs	The Senate Game	Chapter 11 - Planning & Estimating
Steve Hayes	Getting to the right "Pairing" vs. "individual quiet thought"	Chapter 14 - Stop the Maverick
Jeff Canna	I thought I needed more documentation	Chapter 10 - Communication
Nathaniel Talbott	Conversational Pairing	Chapter 14 - Stop the Maverick
Kevin Johnson	Various types of pairing	Chapter 14 - Stop the Maverick
Duff O'Melia	Writing Acceptance Tests before Estimating	Chapter 11 - Planning & Estimating… maybe we need to make a separate chapter for estimating?
Roy & Chris	JAccept description based on XP Universe Paper	Chapter 13 - Write the tests, Run the tests
Rob Mee	Acceptance tests based on a grammar	Chapter 13 - Write the tests, Run the tests
Chris Collins	Unit Testing without	Chapter 13?? - Write the

& Chris Carpenter	Reflection	Tests, Run the Tests
Susan Johnson	When You Can't Be Together (Based on 1999 OOPSLA Paper)	Chapter 31 - Other Stuff
Martin Fowler	Larger XP at Thoughtworks	Chapter 27 - Scaling XP
Laurie Williams	XP Metrics	Chapter 28 - Measuring XP
Michael Feathers	Introducing XP experiences (snippets from his OOPSLA 2000 workshop contribution Extreme Programming – Top Down & Condensed)	Chapters 3-5 - Resistance
Uros Grajfoner & Matevž Rostaher	Programmer On Site	Chapter 19 - Where's The Customer?
Kay Johansen	Experiences Encouraging Extreme Habits	Chapters 3-5 - Resistance
Jack Bolles	Infusing XP into existing project	Chapters 3-5 - Resistance
Ward Cunningham	XP is Genius Friendly	Chapter 5 - Developer Resistance
Joshua Kerievsky	Stuff that has to get done but nobody wants to do (from "XP Flow")	Chapter 25 - Other Roles (Coaching & Tracking)
Joseph Pelrine	Something unique he's done on testing	Chapter 13 - Write the tests, Run the tests
Jeff McKenna	Adjusting to Simple Design	Chapter 17 - Simple Design
Robert Martin	Can XP be used with C++	Chapter 31 - Other Stuff
Ron Jeffries	Why People Go Off Process	?? Chapter 10 - Communication
Steve Wingo	Getting over	Chapter 4 - Manager

Fred George	Management Resistance? Challenges splitting business & technical	Resistance Chapter 11 - Planning & Estimating (Learning Roles)

Forward

[Someone else will write this]

Preface

"You're a fool to think this will ever work."

People have said that to all of us about XP. We've said it to ourselves about XP.

People told Christopher Columbus he was nuts when he wanted to sail west. People told the pilgrims this before they got on the Mayflower. People told many of the early settlers of the American frontier the same thing.

Yet, they all headed west. Why?

They believed there was something better out there, and somebody had to be the first one to find out if they were right.

The journey itself was treacherous for each of them. Once they got to their destination, there were more dangers. Some they suspected ahead of time. Some were total surprises. Of course, they were scared. But, as pioneers, they had no choice but to face their fears head-on. Sometimes they died. Sometimes they lived to face the next life-threatening challenge.

Stories from these brave fools made it back to the people they left behind, and then to people they didn't even know. Those who may not have been brave enough to take the first journey, or who didn't have the opportunity, were encouraged. They became the next wave of pioneers. They were better prepared then the first wave. Bravery and success (mixed with the excitement of knowing the risk they were taking) encouraged another wave. It didn't come easy, but eventually the west was settled.

We are the early pioneers. These are our letters home. We hope they will encourage the next wave to head west.

Introduction

The software industry is in a sorry state, at least from the standpoint of the people who are part of it. Developers hate life. Customers rarely like what they get. The software stinks.

The current environment almost guarantees that software developers, and their customers, will fail. Someone has turned off the light at the end of the tunnel. Disappointment, discouragement, and pessimism are the natural result. It's hard to be an optimist when you never win and you see no signs of that changing.

The Pitiful Programmer

Thomas Hobbes claimed that life in a state of nature is "…solitary, poor, nasty, brutish, and short." With a few notable exceptions, the lives of software developers are the same way. Their most basic needs aren't being met. In a world like that, folks revert to the natural condition of competing for scarce resources just to survive. That's all they can do. They certainly can't thrive.

You may be like the many very intelligent and talented programmers we know who seem to go from one failed project to another. Often, a particular project started out great, but it went south in a hurry.

It became clear that your leadership made unreasonable promises, and set the bar too high. The inevitable technical and organizational obstacles sprang up. Scope crept. Maybe, after Herculean efforts (often at the expense of your family, social life, and physical or mental health), the project actually produced something. Not something to be truly proud of, mind you, but something. Equally likely, the project got cancelled, but that didn't surprise you or anyone else. The warning signs of impending failure were all over the place and screaming for attention.

This is the norm. Being part of a project that provides an enriching learning environment, produces something good, and doesn't kill you along the way is a fantasy, an unattainable dream. You believe success is just luck, and the chances are too slim to count on. The best you can hope for is to survive without being maimed.

The Sad Sponsor

Or you may be like the many business people we know who doubt that they'll ever get quality software delivered on time and within budget. Who can blame you?

You ran the numbers before the project started. You did the due diligence that you were supposed to do. You vetted the numbers with all the right people and there was a unanimous “go.” Everyone was excited about the possibilities. Then reality set in.

Several months into the project, the software team started requesting additional resources in order to hit the target date. After you reluctantly cut the hoped for profit, you found out that they doubted they would ship anywhere close to the time you wanted. And the “minor” functionality changes that some stakeholders requested produced a groundswell of resistance from the developers who got the shakes when you asked their supposedly flexible technology to deliver. They seem to have forgotten that the customer is the one paying their salaries.

You had to ship or you would have blown twice your budget with nothing to show for it. Even if you did ship, there was still a good chance you would get fired for incompetence when you couldn’t even come close to the incremental profit you promised. The next release doesn’t look any better. Is there any chance you can survive in this organization? Even if you can, do you want to stick around with your current reputation?

This is the norm. The best you can hope for is to survive without looking like an idiot.

The Smelly Software

Both the developer and the customer are battered and bloody, if not dead, after the typical project. And the project delivers sub-par software at best. Usually it’s junk that everyone is at least a little bit ashamed of. In private.

Based on the prevailing way of doing things, this shouldn’t be surprising. Scope got out of hand fast and changing requirements invalidated the original design. Pretty soon, nobody even remembered the original design anymore. Under the intense time pressure, developers took all sorts of shortcuts and did all sorts of dumb things. Remember, they’re just trying to survive. Communication broke down, too. Who had time for meetings or coordination?

In the end, the software they created looks something like a 1975 Ford Pinto held together with old speaker wire and duct tape. It’s a time bomb with no resale value. And don’t slam the doors too hard or stuff falls off. It’s full of bugs, or is a patchwork of fixes commented with “Not sure why this works - DON’T TOUCH!” It’s brittle. Changing it is so risky that developers perform unnatural acts in an attempt to squeeze new features into spots not made to accept anything new. That’s the only way to avoid new bugs. Management and customers don’t understand why it seems to be so hard to get the new features in for the next release. Now the pressure is on again.

Developers don't want to produce software like this. Customers don't want to buy and use it either.

How Things Got So Bad

We made our own mess.

Software development as a discipline hasn't been around very long. And it came about almost by accident. In the beginning, it wasn't even seen as a discipline, because there weren't enough people doing it. Then it exploded.

Practitioners cast about looking for some guiding principles to increase professionalism, to improve the quality of the software they were making, and to make life easier. Along the way, both practitioners and theorists with good intentions made some very bad assumptions. These assumptions led to faulty conclusions and bad practice. That made the mess.

The Methodologists' Solution

For years, methodologists have told us that the way to clean up the software mess is to learn from other engineering disciplines, and we have gone along for the ride. Other engineering disciplines would never accept the sorry state of affairs that exists in software. If engineers, or even house builders worked this way they'd be bankrupt in no time flat. Methodologists tell us that we should learn from them.

What do other engineers do? They spend a lot of time gathering requirements to make sure they understand what the customer wants. Next, they figure out the needed raw materials and how to assemble them, culminating in some standard diagram. After reviewing this diagram carefully they approve it and get down to the business of creating something real. The product undergoes rigorous testing and inspections to make sure it's acceptable before any end-customer gets hold of it. They can't afford to miss their estimate by very much, or so the theory goes.

So, the methodologists say, we should build software like civil engineers build bridges, to pick one example. After all, they've been doing what they do for far longer than "software" has even been a word. Using their approach will make us successful. We'd be arrogant and foolish to think it should be done any differently.

As the British say, this is utter tosh. It's bunk. Let that sink in.

Remembering the "Soft" in Software

Software is fundamentally different from physical stuff. It is expected to change. That's why it is called *soft*ware. The stuff that we understand, that we can set in

cement, goes into the hardware. The stuff we don't understand gets left to the software developers.

When we treat software development like a civil engineering project, we sign up for the baggage that comes along with that approach. We have to get all of the diagrams signed off. We have to keep them up to date. We have to go up several layers of management to get permission to put a new feature into a project that's in a code freeze. That's the equivalent of getting permission to put a jackhammer to hardened concrete. It's a lousy way to build software.

Why do we want to apply a process for building stuff with "hard materials" to building something with "soft materials"? Let's stop acting as if there is any real merit in this approach and throw it out! It is not a way to win. It's not even a good way to survive.

We should plan and execute differently, in a way that respects the fundamental differences between soft things and hard things. The civil engineering approach doesn't work for software. It produces brittle software late. That has profound implications for the economics of software.

Building Software Like Bridges: The Dreaded "Cost of Change" Curve

When you build a bridge, you end up with something big that's tough to change and probably will last for over a hundred years, until it decays and collapses. All of these are good characteristics for a structure you trust your life to.

But what if the bridge needed to change significantly next month? What if it needed to change tomorrow?

The common assumption, and common reality, on most software projects is that the cost of changing software rises exponentially over time, just like it would for a bridge. Most software projects, whether or not they are aware of it when they start, are living with what has become the self-fulfilling reality of this "cost of change" curve.

Let's face it. Software projects begin with excitement. It's full of promise. At that point, there are two ways you can go, if the curve is a given. You can ignore it, or you can embrace it. The problem is, neither one works.

You can get by for a while by ignoring the curve. You can produce something quickly this way. You begin to feel invincible. Then, a few months into it, you try to make the first significant change to the software. The super programmers get it to work, but it feels like you're moving slower than you were. Before you know it, you seem to be crawling, and the list of problems is growing faster than you can fix

them. You're climbing the steep part of the curve, and it isn't much fun. The curve was intended to discourage this kind of behavior.

You probably don't want to end up in this predicament. So you embrace the curve and resolve to heed its wisdom. You spend lots of time understanding requirements, drawing pictures, documenting everything under the sun, and getting all the right signoffs. You produce mostly paper in the beginning, but it proves you really thought about everything. Then the requirements change, or you find out you misunderstood something, and the house of cards collapses, but the ship date can't move. You spend more time trying to recover gracefully, but give up in frustration as time pressure grows intense. You decide you'll just get the thing to work and cram in as many "fixes" as you can. Before you know it, what is about to get shipped to the customer has only a vague resemblance to all of the diagrams you drew, and you don't have the energy or desire to go back and update them. Nobody reads them anyway.

You've just cobbled together another 1975 Pinto. Despite your best efforts up front to minimize costs late in the game, the curve poked you in the eye anyway.

When you use a process for building inflexible things like bridges to build things that are supposed to flexible like software, it shouldn't shock anyone that later change costs more. If you ignore the curve, you are doomed to climb it. If you embrace it, you'll end up climbing it anyway. But this reality is a function of the methods you're using. If you use hard methods to produce soft stuff, you will live that curve. Period. When you use soft methods to produce soft stuff, though, that curve doesn't apply anymore. That is the beauty of XP.

XP Flattens the Curve

Change is the reality of software development. You can't anticipate everything, no matter how much you plan. Traditional approaches to software development force you to climb an asymptotic cost of change curve. The only way to produce good software, and to stay sane, is to use a flatter curve.

If you want to win, you've got to flatten the curve and keep it flat. XP focuses on living in the self-fulfilling reality of a flatter curve, and it gives you tools to get there.

Developing software is a challenge, no matter how you go about it. But XP proposes that we apply four values consistently to give ourselves a better chance to succeed:

- Simplicity
- Communication
- Feedback

- Courage

The fundamental premise of XP is that application of these values via consistent principles and practices will flatten the curve. Those principles and practices accept reality. Requirements change. Scope creeps. Two people working together are simply more productive than the same two people working alone. Not every programmer is Einstein. Some things are out of your control. Stuff happens. Accept it and get going.

XP is no silver bullet. There are still forces outside our control (competitors, human fallibility, etc.). But if the curve is flat, you can walk instead of climbing. That way, you can observe the scenery instead of trying not to fall.

XP also isn't just spin on undisciplined hacking. Applying the principles and practices takes discipline. It takes discipline to write tests first, to integrate often, to get all the tests to run before moving on, to pair program. In fact, the one lesson we should draw from "hard" engineering is that discipline is important.

The difference between the discipline needed for XP and the discipline needed for civil software engineering is in the results. XP gets you reliable, flexible software on a daily basis. You can win with that, whether or not you have the diagrams.

When XP is given a fair shot, both programmers and business people enjoy it. And why wouldn't it be? Fighting uphill battles is tiring. The flatter the terrain, the less fatiguing the battle. In fact, it often doesn't even fill like a battle, especially to those who are used to fighting on a slope.

Wagons Ho!

We have seen lots of different ways to develop software, and we've struggled with the curve. XP is the best way we've seen to keep the curve flat. It helps average, ineffective programmers become very effective. It makes excellent programmers phenomenal. It also makes life more fun.

Just like life, software development should be about more than surviving. It should be about living life to the fullest, about succeeding. History has stacked the deck so that survival is the primary concern. XP is a way of thinking and behaving that can help you move past that.

The road isn't easy, although taking the first step often is the hardest part. We've got some arrows in our backs, to be sure, and we took a few wrong turns. The good news is that we made a map. It might not be the only way to get here, or even the best way, but it works. The odds are good you won't die, or have to eat anybody.

Section One: Getting Started

Many people “get” XP. They are impressed by its simplicity. It makes good sense. It is a refreshing departure from the Death Marches they are used to. This section deals with why people don’t do it, even when they are convinced it’s the right thing to do.

Why don’t people start? Usually it is because they don’t have a clue how to begin. After all, this is some radical stuff for many folks. But there can be another reason. Sometimes fear gets in the way.

Since fear is the elephant in the room that nobody wants to talk about, we’ll tackle that one first. Then we’ll move on to how to take your first steps.

Chapter 1

Dealing With Fear

Far better it is to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy much nor suffer much, because they live in the gray twilight that knows neither victory nor defeat.

-- Theodore Roosevelt

Fear is the single biggest reason why people do not try XP. You must take a leap of faith to begin.

In the movie *Indiana Jones and the Last Crusade*, the intrepid Dr. Jones sought the Holy Grail of medieval legend. When he finally found it, the situation got hairy. He had to pass five “tests” before he could claim the prize. He could pass all but one by being athletic or clever. But the third test was absolutely terrifying: The Leap of Faith. He had to step off a ledge into a chasm separating him from the cave where an ancient knight guarded the Grail. He had to trust that there was an invisible path below him that would break his fall.

Scary stuff.

Trying XP is like that. You have to conquer your own fear first. You must step off a ledge onto a path you can’t always see clearly when you start.

In 1998, Ken was convinced it was time to build a "Software Studio". Just a year earlier, he had started his own company. That was a bit scary, but Ken knew enough about consulting and had enough money in the bank as a buffer that he was pretty sure he wouldn't starve. The Studio concept was scarier. It is one thing to have enough faith that God will provide for your own family. It is quite another to have enough faith to believe he will feed the other families working for you.

But if that wasn't scary enough, he had to figure out how to make a "studio model" work. He was a strong believer that a collaborative environment was the way to go. He had been a part of them before and saw that his greatest successes were in these environments. He knew it was the best way to build software and software teams. But how could he sell it to others?

His friend Kent Beck had been pushing eXtreme Programming for a couple of years. Kent was excited by the results he had seen and Ken knew Kent well enough to know that Kent wouldn't hitch his career to something that had no substance. Ken

had also experienced almost all of the practices of XP in one form or another during his career and had always had positive experiences with them. Kent believed that XP was economically a more cost-effective way to build software. Ken needed an economically feasible way to sell his Studio concept.

At OOPSLA '98, Ken felt it all coming together. He had worked with Bruce Anderson and David West in organizing a "Software as a Studio Discipline" Workshop. He then spent most of the rest of the week talking with Kent Beck, Ward Cunningham, Martin Fowler, and Ron Jeffries about a series of XP Books (of which this was supposed to be the 2nd, but that's another story). On the last day of the conference, he saw his friend Bruce Anderson sitting in the lobby and surprised himself as he announced to Bruce, "I'm going to build an Extreme Programming Software Studio™." (He hadn't trademarked it at the time, but that was then and this is now).

Ken is one of those guys who believes that a man needs to do what he says he's going to do. The rest was just implementation details.

(If you know Ken at all, the way to get him really angry is to say that last sentence with any amount of seriousness).

Did Ken believe that doing XP as it was prescribed would work? Hardly. In particular, he was really skeptical about Pair Programming for "all production code". And, at the time, he didn't really have the option as a one person company.

He remembered what Ralph Johnson said when he decided to start DesignFest at OOPSLA several years earlier… (paraphrased). "I don't know how this is going to work, but I've found that all of the significant things I've done were started by picking a direction and just doing it. The details work themselves out. You make some mistakes so you can do it better the next time. But you just have to start by doing it."

With mentors like this, how could he go wrong?

(He should have suspected something when all of these mentors cheered him on as they watched from the sidelines).

Ken often says that there is a fine line between bravery and stupidity. Quite honestly, there are days when he's not sure what side of the line he is on. Nevertheless, he set out to boldly go where no man had gone before.

He would confidently state his case for XP at every opportunity. Given his current clients, his role with them, and the fact that he was a one-person company he had to first convince his clients to XP with him. He often found that he was much braver than they were (or were on a different side of the aforementioned line).

As a software professional, you can see the Grail you're shooting for: a fun way to develop software that delivers fantastic results. So why do software professionals keep using traditional approaches when they rarely produce the results we want, and usually cause a lot of pain on the way to disappointment? Because it helps us cope with fear.

The first client Ken tried to introduce XP to was the founder of a start-up company. The client had hired Ken because he recognized his experience in building Frameworks. He knew Ken had experience he did not have and saw some wisdom in much of Ken's words when he talked about his craft. He needed a framework he was building refined. Once Ken had gained an understanding of what the client was trying to do with the prototyped framework, he proposed building the production version with an XP approach. The client was intrigued.

Writing the tests first sounded interesting. Pair programming sounded interesting… the client and Ken had actually done some of it as they worked through portions of the prototyped framework together. The client valued the collaboration they had and thought that some amount of pair programming might be good. He agreed to start with a pair programming approach to building the production version of the kernel… not because he was convinced that this was the way to build the framework, but because he wanted to bring another developer (Duff) up to speed on the framework and thought that a few weeks of the other developer working with me might get him there… then we could split off and each do our own tasks.

What was produced in three weeks boggled the mind of the client. (You'll read more about the three weeks as we go on). He was initially very pleased, as was his partner and other employee. But, when the dynamic duo suggested that pairing and XP should be continued, the client wasn't so sure.

Although the client recognized Ken's expertise in building frameworks and respected Duff as a developer, he and his partner were confident that they new how to build software and that XP wasn't it. They were fearful of it. To be fair, we didn't do a very good job of introducing it. (Some of the tips we provide in these pages were learned the hard way… we learned lots of ways not to introduce XP).

Instead, they wanted documentation produced so others could understand the framework, and they wanted applications built by other developers. Each person was given their own assignment, and held to the fire to deliver. If we objected, and suggested using XP (which we probably did way too much early on), we were accused of wasting time and working against them (which was possibly true to some extent) instead of just working on what we were told to work on.

When you are scared, you tend to revert to that with which you are comfortable. This is what the client was doing. We didn't do a good job of convincing them that

XP was a safer way. Part of it was that we hadn't taken the time to clearly communicate what we had learned in our three weeks (and afterward). We weren't that confident of it ourselves… we just felt that we were much more productive when we were doing XP than when we were not. Without compelling arguments, it's hard to convince those who are wary. The more wary they are, the harder they are to convince even with compelling arguments.

In this situation, the client wasn't looking to "heavy" methodology, for a solution, just "standard practices" with which they had previously enjoyed some level of success. They felt that following this approach would reduce their risk. Their necks were on the line. They had mortgaged their house to start the business, and they weren't about to blow it by taking an "unproven" approach. Who can blame them?

The formalism, documentation, and process of “heavy” approaches attempt to quell fear in some by supposedly reducing risk. Those who haven't enjoyed some level of success often think that using a methodology created by people with more experience than they have gives them the confidence of doing things “by the book.” ("The authors must have more experience & wisdom, they wrote a best-selling book"). Following predefined, reusable steps lets you slip into the belief that you can connect the dots and get success. Producing lots of reassuring artifacts lets you feel like you are making progress. If you haven't ever had success in building software before, who could blame you for thinking this way?

This is a false sense of security. Following the steps and producing all the documents distracts people from building the software and slows them down. Think about it. When you’re confronting a deadline, do you focus on writing and testing code or on keeping documents up to date? Code wins every time. Having all the other junk in the way actually *increases* risk by building delusion into the process and impeding progress. It’s Mussolini ruining his country, but making the trains run on time.

If you realize this, you get the code written, as our client was trying to do. They felt secure knowing that everyone was writing some of the code that was needed. The only problem was, that much of the code wasn't what was needed. People who were supposed to be building on top of the framework were not doing it well. If they couldn't figure out how to use it, they had no one to work through with it. Much of the code that was produced had a lot of problems. Features of the framework that could offer a lot of leverage were not. Problems in the framework were hacked around because "there wasn't time to figure out the framework". Even when the documentation on the framework was produced, "there wasn't time to read the documentation". When they did read the documentation and found it lacking, "there wasn't time to discuss it"… there was code to write.

A year and a half later we were called on to help rewrite some of the code that was written during that frantic time. The stuff that was originally written in XP fashion mostly stood the test of time. The stuff that was not (including that written by Duff when he was told to go off and develop on his own) needed to be rewritten for a variety of reasons. Most of those reasons could have been avoided if XP was adopter.

This is the ledge you're standing on when you continue to use many traditional approaches to software development whether formal or informal It is crumbling. Choose to stand there, if you're brave. Mediocrity is the best you can hope for if you do. But you have another choice. You can go for the Grail. The only thing that stands in your way is a chasm called the "unknown."

The Leap of Faith

When you realize that the ledge you're standing on isn't as safe as you thought, you can't be comfortable there anymore. You must move, which means you have to take a leap of faith. To do that, you have to confront fears that you probably didn't know you had.

You usually don't have to worry about falling into chasms like Indiana Jones, but you may have other concerns about doing things in new ways. You might look foolish. The odds are good you're at least partially wrong. You might fail. In Ken's case, you might be fired by your client. That looks pretty much like a professional chasm to us.

The scary part is that you can't make the initial leap of faith a little at a time. Once you step off the ledge and you land on the path, you can walk across slowly. But it's all or nothing for that first step.

Fortunately, there is a way to make taking that initial step a little easier.

Bring a Friend

Find a person to take the risk with you. This can be a fellow developer who's up for a good time, a Team Lead who wants to try something new, or a group of people who are tired of doing things the same old way.

You actually can do quite well by applying the principles and practices of XP by yourself. Your confidence and productivity will go up, and you'll be more justifiably proud of what you produce. But XP is a group thing. You can't pair program alone.

You probably are not used to developing software the XP way. It is outrageously fun, but it takes discipline. You will feel stupid often. If you can be "in it together" with one or more other people, you can help each other figure out the puzzles and

get over the rough spots. If you have a pal, a daunting risk can seem more like an adventure.

Ken was blessed to have Duff be his partner in trying out XP. Duff had read the wiki pages (an earlier version of http://c2.com/wiki?ExtremeProgrammingRoadmap) and was eager to try it. Duff is one of those unique individuals who will try just about anything with enthusiasm.

The first week was difficult, but Ken and Duff both wanted to make this work, or at least give it a good college try. Looking back, the #1 thing that made it work was that they dared to try to make it work. How did they start? They just did it.

Just Do It

One of the four key principles of XP is courage. Without this, you're dead on arrival. Beginning the journey takes guts. The other principles will kick in after that.

So find some friends and get moving.

Chapter 2

The Leap of Faith

Do or do not. There is no try.
-- Yoda

> *When starting XP, do the simplest thing that could possibly work. Find a suitable project to start on, use the simplest tools possible, and drive a spike to explore all of the practices.*

One of the fascinating things about XP is that its values and principles aren't really limited to software. They apply to all of human behavior. Nowhere is this more obvious than when you are starting to use XP.

One of the values of XP is to focus on doing the simplest thing that could possibly work. You should think and behave this way when you're thinking about your first XP effort. This is how you can do that:

- Find a trial project that is big enough to let you explore all the practices, but small enough to give you feedback soon
- Make sure you have the materials you need to do the job
- Drive a spike to learn

Find a Target

Once you find a person, or a small group, willing to take risks and simulate an XP project, pick a project to start on.

An acquaintance of Ken's reacted this way to XP after Ken told him the basics and pointed him to Extreme Programming Explained:

> *I took you up on your advice about reading Kent Beck's Extreme Programming Explained book. It's amazing!…I'd really like to see this thing in practice day in and day out.*

This guy obviously "got XP". But he had no clue about what his initial target should be. When Ken followed up to find out what the guy wanted to know, his first question was this:

If you were introducing XP into a company, especially a new company starting up, what practices would you introduce first? Obviously, XP is too big to dump into a company all at once.

That's the wrong way to think about it. The focus shouldn't be on the entire organization. You can introduce XP to a small group on a small project within an organization of any size. Start small and grow from there. That is the simplest thing that could possibly work.

Your first effort shouldn't be as grandiose as making the world safe for democracy. As a matter of fact, biting off too much could make you choke. This is your "proof of concept" for XP in your organization, so start with something small. Ease into XP like a new exercise routine.

You should pick something non-trivial, but not mission-critical. This will let you get used to XP in a relatively "safe" environment. In a nutshell, you'll have the best chance of being successful if you pick something relatively small that you would like to implement, and that you understand how to describe.

Assemble the Right Tools

How many times have you gotten excited about a new hobby or interest, and gone out and bought lots of expensive gear to get started? You probably felt a little foolish once you realized you didn't really need all that stuff to do it right. You probably felt even more idiotic when you learned that the best practitioners often don't use the "best" equipment. Tiger Woods could probably drive a golf ball three hundred yards with an antique club he bought at a yard sale.

You don't need fancy stuff to get started with XP. In fact you need just a few things:

- Index cards for stories
- Someone to play the customer in the Planning Game
- A physical environment where you can pair program
- One computer per pair of programmers (pairing can actually save you money on mice and keyboards)
- An object oriented language to program in
- An xUnit for the language you chose

With this simple set of tools, you can explore all of the practices of XP.

Driving Spikes

The biggest barrier to getting started with XP is that you really don't know what you're doing. The idea may make perfect sense, you may be excited, but the silent challenge of the monitor gives you the shakes.

What if you didn't have to learn it all at once? That's what we in XP like to call a "spike." The goal of a spike to do some useful work that lets you explore all areas of a problem quickly so that you can reduce your risk later on when you try to fill in the details. Try out all the practices of XP to get a feel for them. You might very well produce crap, but you'll learn and you'll get more comfortable with the process.

There are lots of ways to do this, but all of them explore all of the practices.

The Lone Wolf

If you can't find a pal to try XP with you, you have two choices. You can bag it, or you can give it a go yourself. Take heart. History is full of people who had to brave new worlds alone.

Run a miniature Planning Game. If you don't have a real customer handy, play the role yourself. Write a couple of story cards and prioritize them. Then put on your developer hat. Break the stories into really small development tasks of a half-day or less. Estimate the first few you think you should tackle. Then move on to development.

Before Ken hired his first employee, he was full of ideas of what he could do when he wasn't doing work directly for a client. (He's still that way. He's full of ideas. Some of them are actually good). He had a product he had worked on, a graphical drawing editor framework named Drawlets™. It was a Java implementation based on the HotDraw concept originally developed in Smalltalk by Kent Beck and Ward Cunningham in the mid to late 80s and re-developed by many others since then. Ken was familiar with many of the things that had been done with earlier HotDraws and had a bunch of ideas of his own that had not been implemented. He could make a career out of adding features to Drawlets™ if he only had the time, money, and desire.

He looked at his time commitments and realized that he only had the time to get a subset of the feature ideas implemented. So, he started writing them down on cards. He wrote only enough to make it clear to himself as a developer what the "story" was. He played the role of the customer. (Which of these features would make this most attractive?). Once he had those sorted out, he played the role of development. (How long would each of these features take?). When the features were big he laid them aside as a signal to ask the customer to break up the story further if possible.

In less than an hour, he had created a pile of 40 or so stories and had sorted them into piles of high, medium, and low priority. It was clear that he would not have enough time in the next few months to tackle all of the high priority items. He calculated the greatest amount of time he might have to spend on them and realized he would probably only have time to do 2 or 3 in the following two months. He learned an immense amount about the reality of what he could and could not do with Drawlets in the near future, and got a feel for the power of the Planning Game.

Next, determine and write the first tests that will demonstrate you've accomplished the first task. For each test, write the code and run the test until it passes. Once you've gotten a green bar, verify that the test is adequate and that the solution is as simple as possible. Refactor as necessary. Then wrap up development.

We'd strongly recommend doing this one test at a time. If it will take you 3 steps to accomplish the test, write the test for the first step and then get the first step to work. Then write the second test… (more on the approach to writing tests later). The moment you get xUnit to pass the first test, you will get your first taste of the exhilaration of KNOWING your code does what you wanted it to do. It only gets better.

Identify how long the task really took and record how accurate your estimate was. Do whatever sort of version control you need. You now have a baseline for future integration.

Move on to the next task and repeat the process. Estimate. Write tests. Write code and get the tests to pass. Refactor. Record results and do version control. Feel the rhythm. Reflect on what you learned, share it with others, and determine how you could apply it for real on something mission-critical.

Congratulations. You've just done XP. If no one was willing to join you, start doing things this way on your own. When you start producing results that others can only dream about, XP will catch on.

A Single Pair

This approach looks almost the same as that of a single programmer. The difference is that you write tests and code as a pair. You should take turns being the "driver" of the pair (the one typing code) and the "passenger" (the one thinking ahead).

A Small Team

This is basically the same as with a single pair, but you will get a fuller feel for what XP is really like. If at all possible, start this way.

When the team is running the Planning Game, have each person take ownership of and estimate a couple of tasks that seem most important to the group.

Once the group has completed the Planning Game, discuss ideas for how you think you should describe the main objects in the system. This is a skeleton of a System Metaphor. Don't worry about getting it exactly right, just agree on something.

Have a stand-up meeting to decide which tasks should be tackled first and how you should pair. Then pair up to write tests and code, switching pairs as often as you can. During development, integrate the versions of all tasks on a single machine, making sure you can run all tests each time you integrate. Resolve any conflicts with the existing integration base each time you integrate. Whenever a pair integrates, don't allow them to continue until they have integrated cleanly.

A Small Team with a Lead Developer

This is a variant on the previous approach, where you have a lead developer who is more confident in his abilities to pull this off. Perhaps he has some XP experience already. In any case, this lead developer brings a less confident team along. He follows the steps above, demonstrating a simple version of each and then saying "now, you try it in pairs." The effect is that he is a developer working on a task with multiple development partners…the team is his "pair."

This last approach was how we started out at our largest (and first) XP client to date. They were looking to bring a strong Java development team in to build their next generation software product. They wanted to bring in a team of experienced Java developers to work with their small team who was new to Java and bring in the best practices. Ken convinced them that XP would not only be the best practices for the long run, but the best way to work with their people to transfer the knowledge.

We taught them XP and Java at the same time. Day one we talked about XP and wrote our first JUnit test before there eyes. It was incredibly simple. Ken had his laptop attached to an LCD projector and asked them to tell him something they might need to do in their new system. They said they would need to collect "results" from their hardware device and associate them with an "Order". So, Ken wrote a test that added two results (objects that didn't exist) to an order (another object that didn't exist) and tested that when the order was asked how many results it had, it would answer two. He then wrote just enough class and method stubs to get it to compile and ran the tests. A red bar appeared. He then implemented the "add" method and ran the test. Red bar. Next he implemented the "getNumberOfResultsMethod". He ran the tests. A green bar. Applause from the other developers in the room. By the end of the day, we had seven tests running.

The next day, we discussed some simple things we might need to add to our baseline and told people to go off in pairs. Over the next couple of hours, each pair had written tests and gotten them to pass… sometimes with help from Ken. We went back to the conference room with the LCD several times, each time discussing

another type of feature we wanted to add (UI, interfacing to serial port, printing), sometimes adding a stub for some tests, and then going off into pairs.

At the end of the first week, we did a little planning game on a prototype of the new system. Then we were off.

It's All Right To Feel Awkward

When you first try XP, most of the practices will feel funny. You will write tests before you write any code. You will be coding with a pair all the time. You won't flesh out all the details before you start writing code. These things mostly likely are (really, they just appear to be) vastly different from the way you have been doing things for years. It's normal to feel like a fish out of water when you start doing things differently.

When Walt Disney was a young teenager, he saw an urgent ad for a trombone player in parade that was two days away. One of the trombone players in the band was sick and wouldn't be able to play. The bandmaster was distraught. Disney introduced himself and volunteered for the job. The bandmaster was ecstatic. He told Disney to show up at 7am sharp in two days to pick up his loaner trombone and line up with the band.

On parade day, Disney took his place in the brass section. Just a few minutes into the parade, the bandmaster heard a sickly warble from the area of the trombones. It never got any better. When the parade finally ended, the bandmaster ran to Disney and screamed, "Why didn't you tell me you couldn't play the trombone?" Disney replied, "I didn't know I couldn't. I never tried before."

That's the attitude you need when you take your first steps with XP. It will feel funny. It will be a little scary. You won't know if you can do it until you try. That's the leap of faith you have to take.

Chapter 3

Taming the Resistance

Great spirits have always encountered violent opposition from mediocre minds.
-- Albert Einstein

> *XP forces people out of their comfort zones. They will resist. This resistance comes from fear and pride. Overcome this by focusing on using XP as a strategy to increase their chances of winning.*

If you think your own fear and ignorance are the only obstacles you'll face when starting XP, think again. That is just the beginning.

Other people will resist. They can't afford not to. They have a good bit of time, effort, and ego invested in the way things have always been done. Change might be risky, painful, or both. It also might cast doubt on their judgment in the past. You are pushing them out of their comfort zone, and they will not like that idea.

You'll get resistance from two primary sources:

- Managers
- Developers

You probably won't be shot. Beyond that, all bets are off. We've seen everything from reasonable and open debate to screaming matches and sabotage. It can get ugly.

The most likely form of resistance will be simple objections that XP is wrong, stupid, or inferior for one reason or another. These objections supposedly are based on principle. Most of the time, they aren't.

Where Resistance Comes From

Managers and developers are people. We human beings are wired to fear the unknown and to think we are worth more than we are. If you don't believe this, you haven't been paying attention. Unfortunately, both of these natural behaviors can cause problems.

When we are afraid, we gravitate toward ways of thinking and acting that make us feel safe. This usually means we fall back into our old habits, even if they are

unhealthy or unproductive. New ways of doing things can be scary. This means that people will tend to slip back into the old ways of doing things. This inertia is natural.

It is unhealthy to think you are worthless. But thinking we are worth more than we are vaults us beyond self-esteem to pride. The only way we can learn is to admit that we don't know. They only way we can change behavior for the better is to admit that our current behavior might be wrong. Pride prevents both.

Under stress, we will do what comes naturally. We will be scared and proud.

Managers and developers possess these qualities in different proportion, but they are always there, getting in the way. Not all the resistance you get will from these groups will come up front. In fact, it might take a while to build up a head of steam. It will come eventually, and you have to be ready for it. Fortunately there is a simple strategy that seems to work most of the time. Focus on results.

The Result That Matters

In 1968, a lanky guy named Dick Fosbury revolutionized the sport of high jumping with a technique that became known as the Fosbury Flop. Instead of going face-first over the bar and scissor-kicking his legs like everybody else, he "flopped" over on his back and landed on his shoulders. It looked stupid, frankly. Lots of people said, "We've never done things this way" and "That'll never work." But at the 1968 Olympic games, Fosbury cleared every height up to 7'3¼" without a miss. He won the gold, and set a new Olympic record at 7'4¼" when all of the other competitors failed. It is hard to knock the winner for using an unorthodox strategy.

XP is not the goal. It is a means to an end. Winning is the goal. In the end, it is the only result that matters. If doing something new and different (be it "flopping" or XP) will increase your chances of winning, then that is the smart thing to do.

When you face resistance from managers and developers, be sure they understand that the reason you are talking about XP at all is that you are trying to help your team play to win. Each group will have their own particular objections (we'll talk about these in the next couple of chapters), but they all want to win as they define it. If you can't get people to agree that winning is the goal, it might be time to change jobs. Losing is a hard habit to break.

Will It Help Us Win?

If you can get everyone to agree that winning is what's important, cast XP with each group as best the way to do that. Focus on the results of XP that can help managers and developers win as they define it. This will do two things:

1. It will focus attention on what is really important all the time.

2. It will give you a standard to use when evaluating each practice against other options.

Approach every discussion by asking whether or not doing things the "old" way will make winning more likely. Then listen. Get everything out in the open. Discuss the alternatives in a bunch of different contexts. Make the strongest possible case for the old way. Then consider the possibility that using the XP approach might be best, even if it goes against the gut reaction of most people. Finally, work hard to convince people that the best way to prove whether or not XP produces better results is to try it for a while. Anything else will be merely guessing about how things map from other people's experiences in different environments.

If XP produces better results sooner and with less pain than the old way, there is no contest. If XP produces better results, it will be a tough sell to say that the team should go with the old way just because it is the devil that they know. If the powers that be still stubbornly refuse, you may have found another reason to leave the organization.

What Not To Do

Managers and developers will object to particular practices. You will be tempted to defend them, or your support of them, on theoretical grounds. Don't. These things are irrelevant. Instead, discuss the objection in the context of winning. Remember that winning is the goal, not the individual practices *per se*, or being right. The practices are simply ways to help you win. They are not "good" or "bad"; they are only better or worse then the available alternatives.

The reality is that you will have to justify and defend the practices eventually. If everyone involved agrees that winning is the goal, you have a ready-made standard for evaluating them against other options. If a practice increases your chances of winning, do it. If it doesn't, scrap it or change it so that it does in your environment.

Finally, don't focus on XP for the sake of XP. This will label you a "zealot" in a non-productive religious war. Ken and Duff learned this the hard way at the first client Ken introduced to XP.

Remember that winning is the goal, not XP or you being right. (Even though we were right).

Chapter 4

Manager Resistance

Managers will resist XP because they make faulty assumptions about alternative approaches. Focus on flattening the software development cost curve to overcome their objections.

Most managers got where they are because they have at least some ability to think ahead (we'll assume the best here). They are supposed to see issues before they become issues. Manager objections to XP relate to what they see as potential issues with the approach. Fortunately for you, the objections usually are based on faulty assumptions about alternatives.

The Manager Perspective on Winning

In our experience, there is nothing that overcomes manager resistance like undeniable progress toward the goals he will be measured by. Find out what these are and how you can use XP to help him make progress toward these goals. That is how a manager defines "winning.".

One technique that Ken has used with great success is to start every project by asking the manager, "What can I do to help you achieve your personal goals?". In this conversation, you find out what these goals are and you start off on the right foot with the manager. Then as you present ideas, tie them to the manager's goals. If you can't figure out how to tie them to the goals, don't introduce them.

This can also work in the middle of a project when there is tension between manager and developer. Ask for some of the manager's time. He might be expecting you to dump a bombshell on him. When you start the conversation with, "I realize I may have been inadvertently working against you and that's not good for any of us. Please forgive me. I'd like to start with a clean slate. Help me understand your personal goals so I can figure out how we can work together to achieve them." Once the manager picks his jaw up off the floor, listen to them.

Every organization is different. That means every manager will be measured by different standards. But all managers in all organizations are concerned with one primary objective. They are on a mission to minimize risk.

Remember the asymptotic software development cost curve? Managers are consumed by minimizing the risks to their projects and to their careers implied by that curve. They don't want to get fired and they don't want to end up with egg on their faces. Their knee-jerk reaction to new approaches tends to be that they will cost too much and take too long, both of which increase project risk. When managers object to XP, risk avoidance is behind it. If you want to be heard, you had better couch your responses in those terms. Focus on these things:

- The money costs of code development and modification
- The time cost of code development and modification
- Retention of highly skilled developers

These are all synonyms for risk. All of them make the curve asymptotic. If a given approach does a better job of flattening that curve, it wins.

Here are the most prevalent objections we've heard and how you can address each by focusing on flattening the curve. The proposed way to address these issues here are a bit terse. Hopefully the rest of the book will help reinforce the points made.

XP Is Too New To Trust

Almost nobody makes a habit of being the sucker to use Release 1.0 of a software product. This increases your project risk. To many, XP is at Release 1.0 right now. Maybe it is really at Release 1.x. So what?

When managers talk about new things like XP being risky, they are assuming that the status quo is safer. We have seen that it isn't. If you stay on the status quo curve, be ready to run out of oxygen. You cannot stay there and survive. The only way to find out if XP is too risky to use is to try it. As we said in Chapter 2, you don't have to try it on a mission-critical project first. But you have to try it so that you can get feedback on whether or not it produces the results you want in your environment.

If your manager doesn't micromanage, get a bunch of XP-friendly developers together and give it a try. Show your manager the results. If the results are great, you're on your way. If not, drop back and figure out why not. Then modify the process as necessary and try again.

If your manager is more hands-on, ask your manger for permission to try XP for two or three weeks. Then let them examine the results after a couple weeks and draw their own conclusions. Ask them if extrapolating the results you saw on this first project would flatten the curve. If it would, you've got a powerful argument in favor of XP.

XP Is Simplistic

Managers might say that XP glosses over complexity, or that it replaces thinking ahead with simplistic naiveté.

They are assuming that your existing approaches handle complexity better and let you think far enough ahead. Most approaches we have used other than XP simply don't. They deal with complexity by trying to figure it all out up front. You draw pictures, create documents, try to cover all the bases and anticipate everything. The problem is that you're guessing. Pretty pictures are no substitute for experience.

XP is simple, not simplistic. It recognizes the reality that things change. You cannot predict the future with any degree of accuracy. You cannot plan for everything. You might guess right, but that's all it is. XP says, "Don't even try." Instead, do things in a way that lets you respond quickly and well to emergent reality. What does that look like?

The Planning Game helps to identify items with the biggest risk. We tackle these first. Planning is so important that we do it daily, based on real experience and a working system, not once based on our best guess when all we have is theory. Two sets of eyes see all production code, which minimizes the number of sides that are "blind." Because there is no code ownership, and we have tests for everything, we are able to move our best developers to the problem areas that need their expertise, without leaving the rest of the project in the lurch. And so on.

XP forces a team to find real problems sooner, without having to guess at what they are. It reduces the number of blind spots. Its rapid feedback allows it to be simple without being simplistic.

Before assuming too much, ask the manager why he thinks it is simplistic, and what additional practices he thinks are necessary to make up for the simplicity. If they can supply them, ask what it is about those practices will help you accomplish that the other twelve practices do not. In particular as why they think it will get him the results he wants faster. Often it is a misunderstanding of what the practices are.

We find the biggest objections in this category are that more documentation is needed to help current and future developers understand the system. See the sections on Communication and Testing to provide fodder for your argument.

Pair Programming Is Too Expensive

"We can't afford to pay two people to work on the same thing!" This is pure money risk. Fortunately, it just ain't so.

This objection assumes that pair programming costs more than having people program alone. Research has shown that "development time" does increase with XP by about 15% (not 100% like a knee-jerk reaction might suggest). However, these

costs are offset by higher design quality, reduced defects, reduced staffing risk, increased technical skill, and improved team communication and morale.[1] It is economically more feasible to program in pairs.

Our personal experience tells us that the economic feasibility goes up as the group size increases, assuming the other practices of XP are in place. This is because the overhead of communication is proportionately much lower while the quality of communication goes up (see the section "I heard it through the pairvine".

Even if it were more costly in terms of development time, it is still better to program in pairs. The toughest part of programming is design, and that happens at the keyboard, not in an architecture picture. Would you trust your design to one person? If you don't pair, you are. You are building extreme dependence on your "stars". Even worse, you are counting on your less experienced developers to implement that design without significant checks on their work.

Without pair programming, you are opening the door for more bugs to make it into later testing and into the released product. You are meeting all the prerequisites to attain hard to maintain code. Is it better to have someone else verifying the code is up to par as it is written or to depend on unchecked code? Suppose you actually produced the documentation of this code at some point. Why is it too expensive to have a second set of eyes on the code but not too expensive to have a developer take the time produce unverified documentation of the more than likely subpar code? How can you be sure that the documentation (even for some moment in time) accurately represents what is in the code? If it is generated from the code to keep it up to date, it doesn't tell you anything the code can't tell you, so why do it? Ask the manager to be as scrutinizing about existing practices as he is about pair programming. If you can get him to be honest, he will quickly discover that traditional practices increases his project risk to the point that he cannot afford not to pair.

You might not pair all the time. There are tasks (such as exploration or debugging) that might make sense to do alone sometimes. But pairing should be the norm. Anything else is too risky.

I Can't Afford A Full-Time On-Site Customer

If you want to move at full speed, you can't afford not to have a customer on-site full-time to answer questions for the development team. We don't know of any

[1] From a paper by Alistair Cockburn and Laurie Williams called *The Costs and Benefits of Pair Programming*. You can find a link to it at http://www.pairprogramming.com, along with other good stuff. Refer your manager to this paper. Better yet, give your manager a copy.

research that's been done in this area, but we've been on many projects. When developers have to wait for answers, two things tend to happen:

1. The project slows way down or grinds to halt while they wait, or
2. They get tired of waiting and they guess at what the requirements are.

The first result costs you money by delaying the realization of value from the project. The second costs you money, too. Developers often guess wrong, and have to redo lots of work. That produces still more delay.

Ask yourself a basic question: do you want the developers to give you the software they assume you need? We suggest you save yourself the financial and psychological pain. Put a customer at the disposal of the development team.

Better yet, ask another question. What is the alternative? A study[2] recently showed that the typical requirements document is 15% complete and 7% accurate. That stinks! If we rely on the requirements spec, we will fail. Suppose we accurately guess how to fill the holes 50% of the time. OK, now we are up to 57.5% complete and 53.5% accurate. Do you feel better yet?

It is certainly true that it will often be difficult to have a customer on site or available all of the time. Is that any reason to ignore the problem? If we do the rest of XP, are we any worse off? We still are guessing to fill the holes.

XP Is Too Informal

Managers may take issue with XP not having lots of documentation. They may claim that XP is an excuse to code like a cowboy – it's hacking without documenting. They might also think that Stories aren't formal enough to give programmers enough information.

The most common source of the informality objection is a bad experience in the past. Most managers have been burned before. They have no desire to repeat the unpleasant experience. But they throw the baby out with the bathwater. They assume that formality will increase their chances of winning. Nothing could be further from the truth.

Lots of documents do not make software better. Clearer code does. The only way to keep code clean is not to clutter it up with unnecessary stuff, and to scrub as you go. XP lets you concentrate on doing both. Build only what the Stories for the current iteration call for. Refactor mercilessly.

Using Stories instead of detailed specs is no excuse for being lazy. We still have to get all of the requirements. We just don't need all of the details for ones that

[2] Need to get the exact reference from Highsmith talk

won't be implemented for a while. Experience tells us that these will be completely different then anyway. Remember that a Story is a commitment to have a future conversation to flesh out the details. You need to verify the details of any Story before you implement it, so we defer the detail gathering until then. XP requires that you produce all *necessary* documentation, but discourages production of unnecessary wallpaper.

By the way, why is an approach that encourages us to write tests before we code less formal than one that makes us write documentation before we code? It's often important to question the value of formality and focus on what we really want… results. If the formality we are familiar with gets us the results we want, why change? The only reason to change is if the informality gets us the results we want faster, cheaper, and/or more reliably. Call us crazy, but we'd go with running tests over paper documentation anyday when we are looking for reliability. If reliability doesn't matter, then all that's left is faster and cheaper. We're going out on a limb here, but we'd venture to say that producing formal documentation does not make the process faster or cheaper.

Be Wary of "XP-Lite"

Sometimes there is no formal resistance from a manager. They simply state boldly that XP sounds interesting, but that the organization can't afford to try it now. "Maybe after we transition the wumbubbits to the hinklefarb platform," or some other future milestone.

At this point, you will be tempted to pick a few XP practices and implement them, rather than the whole ball of wax. There is nothing inherently wrong with this. Maybe having more tests, or programming in pairs, or playing the Planning Game will help flatten the curve you're currently on. Be careful.

Bringing in a single practice that is most likely to have an immediate positive impact can make people more receptive to the next one. On the other hand, it might make people think XP is just a bag of practices that can be chosen at random. We have seen that XP is much more than the sum of its parts. All of the pieces work together to produce something amazing.

Many have had success introducing XP one practice at a time. We recommend that you introduce it in "chunks." Introduce at least several practices at a time, and don't let more than a week or two pass before introducing the next chunk. If you do it this way, people will recognize how each of the practices supports the others. If you do it one at a time, your project could be way out of balance.

You should certainly be smart about it. The end of a project might not be the best time to introduce a full-blown planning game. You probably don't want to pair your database guru up with someone who can't spell database a week before a

deliverable. You certainly don't want to refactor too much without first having a critical mass of tests around the stuff you are refactoring. But, we can't think of a time during development where writing a unit test before I write new code would be a bad thing. We don't know when the customer shouldn't be in charge of setting priorities.

If you are satisfied with having your process "stink less" and not being as far from the best as you used to be, introduce a new practice every year. If you want to play to win, don't play around.

Chapter 5

Developer Resistance

Developers will resist XP because they might not see the advantages. Focus on the environment fostered by XP to overcome developer objections.

Developers Are Different

Developers are born with a "geek gene." They are certainly different from managers. Some can play both roles well and shift transparently between them, but those people are rare. Developers love to write code. It's why they get up in the morning. When they can do it better than others, when they can know more, they are rewarded. They derive their security from knowledge and ability.

People drawn to software development (like the authors) tend to be people who would rather communicate with a machine than with people. Most of them were labeled "smart" at a young age. They spend years in school learning lots of details about how to tell machines what to do, which the rest of the world find about as entertaining as watching grass grow. They complete programming assignments alone, since getting help would be cheating. When they graduate, they get paid big salaries. After a few months on the job, managers realize that developers don't play well with other people. Duh.

XP rebels against deeply ingrained programmer habits by forcing these folks to interact with people most of the time. They resist.

The Developer Perspective On Winning

In our experience, there is nothing that overcomes developer resistance like focusing on the environment fostered by XP. Use the geek gene to your advantage. Developers care about stuff like this:

- They want to develop software, not write documentation or spend all day in meetings.
- They want to be proud of the software they develop.

- They want to have fun doing their jobs, not feel like they are undergoing surgery without anesthetic.
- Once the software is “released,” they don’t want to get stuck maintaining it forever, or go through hell to change it when they have to. They want new challenges, not just the same old challenge of having to figure out what unnatural act they can perform to patch the old code.

These are the opposite of what is true when you have a steep cost curve. Unlike managers, developers don’t particularly care about a flat cost curve. They do care about the pain associated with a steep one. Having an environment that avoids that pain is how developers define “winning.” If you want to be heard, you had better couch your responses in those terms. If a given approach does a better job of producing that environment, it beats all comers.

Here are the most common objections and how you can address each by focusing on results that matter to developers. Again, the proposed ways to address these issues here are a bit terse. Hopefully the rest of the book will help reinforce the points made.

XP Is Too Simplistic

Sound familiar? Managers said the same thing, but for a different reason.

XP challenges the perception of developers that no one understands what they do and can’t do it themselves. This is true to some degree. But it shouldn’t be true because of the process they use. XP takes the magic out of process and lets it live in the code, where it belongs. The magic is in the results.

XP gets out of a developer’s way. You produce no unnecessary documentation. You get to focus on designing and developing good code. You get to enjoy your time, rather than loathing having to come to work.

XP is not simplistic. It is simple, uncluttered. Some aspects of the discipline aren’t easy for developers (see the discussion of pair programming below), but it has the best chance of any approach we have ever seen of fostering a rich environment for programmers.

Often, when an experienced developer says this what they really mean is "I've learned other ways to do software (which involves little programming) and my career has advanced because of it. I don't want to go backwards (at least not career-wise). And besides, I'm not as good at programming as I once was because I spend most of my time doing *higher level* things." Very few of these people actually think the higher level things they do are as fun or that the process they use to do those higher level things are efficient. They are often scared that what they've gotten good at will be looked at as less valuable.

What you need to point out is that many of those skills are still needed. Their ability to do "architecture" is still needed. They need to help find a Metaphor and make sure people are focused on how the pieces fit together. They just need to do more of it via verbal communication and interaction. Instead of being frustrated that people don't understand their architecture and that they don't have time to make it work, their time will be freed up to spend more time communicating the architectural issues and assisting the other developers. As a bonus, their hands-on technology skills will increase.

I Won't Like Pair Programming

There are two things behind this objection. First, it is uncomfortable for developers to work with others. They've never done it before. Programming is perceived as a lone wolf activity; XP says you should hunt in packs. Second, whether they want to admit it or not, developers aren't particularly fond of giving up their place in the sun. With no code ownership, they believe it is harder to be the hero. This is the developer pride problem rearing its ugly head.

The way to handle this objection is to focus on the benefits of pair programming. Emphasize the knowledge sharing, the improved code quality, the increased morale. Then suggest an experiment where you do all programming in pairs for a couple weeks. At the end of the experiment, you can decide as a group when it makes sense not to pair. We've found that programmers start to like pairing, and see the benefits, after about the first week. They will start to communicate better and to feel more human. During the second week, they normally start to hit their stride.

In a study Laurie Williams conducted, she found that 90% of people who tried pair programming liked it. Our guess is that some of the other 10% could be won over with a little more sensitivity to identifying what it was about the experience they didn't like and making some adjustments before a second attempt is made.

When the trial period is over, ask them what parts of pair programming they are still struggling with. After they have shared, give them a simple choice. You can work through those issues as a team, or they can "give up." Be sure to put it this way. It is both truthful and motivational. Developers hate to quit on a problem.

You might face a "green eggs and ham" team. They might refuse to try pairing because they are sure will hate it. If no one on the team is willing to try pairing, you've got two choices. You might determine that you work with people who are afraid of trying anything new and decide how long you want to stay in this environment. Or, if you have some sort of authority, you might find a creative way to force them to try it. For example, decree that the most critical portions of the system must be developed in pairs. This will make pairing more attractive. State that

any code written without a pair will be subject to formal code reviews. Developers hate being subject to those, and they hate doing them even though it has proven to be a huge contributor to software quality[1].

There have been several studies that show Code Reviews are the biggest contributor to software quality. Even above testing. This is amazing considering how poorly we execute Code Reviews. A bunch of people who haven't struggled with the problem get to pick apart those that have. Certainly we are supposed to provide only constructive criticism, but it is very difficult for the person having their code reviewed to not be on the defensive and it is difficult for the "gurus" in the room to not find something to pick on (to establish that they are smarter than everyone else).

One study showed that 90% of the benefit of a code review was attained by the first person reviewing the code. We think this is due to two main reasons: 1) before you present your code for review, you conscientiously clean it up so you don't look bad, and 2) most of the things you missed because you were so engrossed in the problem you were trying to solve, any other competent developer would see as soon as they looked at it with a fresh, critical eye.

Among other things, pair programming get you these benefits in a less intimidating and more productive way. You don't have to do as much "code cleanup" because you are always conscious that your work is being viewed by someone else. You don't have to be as defensive, because the constructive criticisms come a little at a time. When you just "hack something to see if it will work" you can communicate that this is what you are doing to your pair. Your pair will then remind you that you have to go back and clean it up. And, of course you both get to learn from each other and have discussions about the value of certain approaches as peers rather than as someone who is being interrogated.

Pairing is critical to success. It also is a healthier way to develop software. This is a case where you might have to practice a little tough love. It is for your developers' own good.

XP Is Weird

We are the first to admit that some parts of XP sound a bit odd to "classically trained" (or "classically untrained") programmers. These tend to have the greatest shock value:

- You write tests before you write code to exercise the tests
- You let the design evolve to fit changing requirements

[1] Need to hunt down some of those studies that have shown Code Reviews shine.

- You program in pairs, often sharing a single keyboard and mouse

If you're like us, you didn't cut your teeth in an environment like this. It takes some getting used to. The way to handle this objection is to focus on the freedom XP can give developers to concentrate on writing code. That is what they love to do, after all.

Writing tests before they write code feels weird at first, but it is similar to what they've always done. When you write code, you start with some functional requirement and then map that to code somehow. The code is in your head for at least an instant before you write it down. Stop! Now write a test that will verify what you expect that code to do. Then write the code to make the test work. When most developers stop to reflect, they will find that doing things this way forces them to think about design and then to verify it quickly. It may seem that it slows you down at first. And, when you are first figuring out how to write tests it probably does. However it doesn't take long to see the benefits. Work through it!

The results are phenomenal. Your code will be simpler, since you only had to write enough of it to get your tests to pass. You don't have to be afraid of other people changing your code, since they can run the tests for validation. Having tests gives your manager confidence that he can let you move on to greener pastures. And tests also serve as good documentation, so you're killing many birds with one stone.

The best way we've seen is to start writing the tests yourself even while the other developers refuse. Every time you experience one of those "Thank God, I had the tests" moments (trust us, it won't take long), talk about it. Pretty soon you will be test-infected[2]. The infection will spread.

Letting design float with changing requirements feels odd, too. People ask us often if we're nuts. No, we're not. (Well, at least not due to this issue). We're realistic. Your design will need to change whether you accept that reality or not. XP just makes that normal. You won't let design fall on the floor, you'll simply spread it out. Rather than doing it all up front, you'll do it "just in time."

Again, the results of incremental design are amazing. Changes in requirements aren't cause for mass suicide. Your designs will reflect reality instead of theory that you guessed about months ago before you knew what you know now. Designing this way produces code you can be proud of, not a jalopy you cobbled together.

On the other hand, this is not an excuse to do no design! As soon as you realize that more than one object is calling the database, and that embedding database connectivity into your code is making it "complex" (which may or may not be before you write a line of code, depending on your experience), it is time to consider seperating a database layer from your domain layer. We've seen the benefit on one

[2] Reference K. Beck & E.Gamma article (Java Report?)

project where the database was switched on us twice due to business negotiations gone bad. Due to our good design, switching the database out (and adjusting the persistence layer) could be done without having to change many of the other objects. The point is that we didn't need to spend months designing the persistence layer to figure that out.

On another project, we needed to convert one key object to XML. We thought, "What is the simplest thing that could possibly work?". One of the developers did a quick web search and found a free product that would automatically do the conversion. However, it was fairly large and we'd have to figure out how to use it. Then someone noted that converting that relatively simple object into XML could be done in a single method without this product. Done. If the requirements change and we need to start converting more complex objects, or a lot more of them, we'll revisit our approach and this product. Until then, we are moving on to other stories.

The bottom line is that feeling awkward is normal when you're doing something new. Think about it like learning to ride a bike. When you started you stunk at, and probably fell off once or twice. After you got the hang of it, it felt natural.

XP Doesn't Give Us Enough Information

Developers don't like to create a lot of documentation (or any, we suppose), but they do like to have enough information to act on. They don't want to fly blind. Nothing hurts quite as badly as working like a dog to deliver something nobody likes. We hear often that XP sets you up for failure by not having enough design documentation, and by depending on stories instead of comprehensive specs.

Ken and Roy have worked on lots of projects. In most cases, your first day follows the same pattern. You walk in and somebody hands you a functional spec, a technical spec, and/or a detailed design document. Nobody ever reads them and they're hopelessly out of date, you're told, but here they are. This is no better than having no documentation at all. Often, it's worse.

Several years ago (pre-XP), Ken went to help a group who had lost a developer due to a family emergency. He was told that, when he got there, they were a day or two away from delivering "Phase 1", a functional prototype of a subset of the system. Since the chances were that he couldn't help finish that, (by the time he got up to speed, they'd be done), he should begin thinking about the next version of the system. Being an agreeable sort, Ken said, "fine, how would you suggest I get the background I need without getting in your way?". "Read these documents. They are fairly complete, but mostly out of date. Section 6 gives a pretty decent account of the motivation for what we want to do. The prototype doesn't follow much of the implied design, but it works and is probably a bit more thorough."

So, Ken spent close to two days working through the documents. There wasn't much design to speak of (at least nothing that could be reasonably implemented), and Section 6 was full of acronyms and phrases that were not in Ken's vocabulary. There was a lot more to read, but at a cursory glance Ken didn't think he would get any more benefit from reading it (other than the bliss that comes at the end of the day when you get to stop reading), so he asked if he might review the prototype to help him understand how they had approached the problem for the prototype and begin to make recommendations of what approaches they might keep and what approaches should change. Permission was granted since it turned out that Phase 1 still wasn't over due to a lot of bugs in the prototype that the rest of the team was working hard to get out.

As he studied the prototype, he discovered that it was built on top of a framework that another group had built for a different application. Although there was little documentation on that framework, an occasional class comment explained some of the trickier parts and the code was clear enough that he could figure out the rest.

On day three of reviewing the prototype and doing some experiments to explore how they might approach things differently (day 5 of being there… 4 days over the day Phase 1 was supposed to be complete), Ken overheard the morning panic meeting that happened to be held in the middle of his cube set. (Each developer had their own corner of the square, and a small round conference table was in the middle. The other 3 developers were working on finishing Phase 1). One of the developers was sharing with the manager that the bug they were discussing was a bear and that it would be nice if he could do something like X-Y-Z, but that was going to take some time. Ken interrupted, "Excuse me, Bob, but I couldn't help overhearing. There is a feature of the framework that already does X-Y-Z." "Really?". "Well, at least I think so. After you're done the meeting, I'll look at it with you and we can make sure it helps you do what I think I heard you say." "Cool. (turning back to the manager) Well, if Ken's right, then we can knock that one out today and I can look at some of the other bugs that should be easier."

Ken was right, and the next day at the "we just gotta get the last bugs out of this thing and we'll be done with it" meeting a few more hairy bugs came up. This time, Ken was called on. "Hey, Ken, you wouldn't happen to know if there is anything in the framework that will let me U-V-W, do you?". "Actually, I think there is something that will do U-V for you. And the W part is a piece of cake.".

What's the point of the story?

All the developer's had read the documentation and realized it wasn't that helpful so they threw something together anyway. They were fortunate enough to have some decent code from the other group to build upon, but they never "read the code". The documentation was what was supposed to give them understanding.

Since the framework they were sitting on didn't have much documentation, they never tried to understand it. Somebody showed them how to do something with it, and they ran forward. If they would have taken two days to read the code instead of reading the documentation, they probably would have been done Phase 1. The code was up to date and it provided something useful to study. The documentation provided almost no value. It certainly provided less payback than the months of time and money that was put into producing it.

When Ken asked the "Phase 1" developers how they built the prototype based on the information in the document, he was told that they didn't. They had read the document and still didn't know what to do. So, they started meeting with the users who explained to them what they really wanted, and built the prototype in between meetings based on what they learned during the previous meetings.

XP does NOT recommend that you don't produce documentation. It just says you should not spend time creating any unnecessary documentation. You need to figure out what that is. Our recommendation is to throw out pre-conceived notions of what is necessary and assume you don't need it. Then, when somebody asks for it, ask them why they need it. If they "need to understand something", offer to explain it to them. If they still need it, consider it. See the section on Communication, Not Documentation.

Unless there is some external requirement to do so (e.g. in a regulated environment), we do the simplest thing that could possibly work for documentation:

- Stories, tasks, the developer-customer discussions around them, and the acceptance tests are the only spec you'll ever need.
- The "system documentation" is mostly in the code and the unit tests which always tell you how the code works today.

Recently, we were in a meeting with a prospective client. Part of the conversation went like this:

Client: We have this software that was written several years ago by someone who is no longer with the company. We brought in a new person to try to add some new functionality and he's having a rough time.

Ken: Do you have out of date design documentation?

Client: Yes.

Ken: Would you rather have a set of working tests?

Client: Absolutely.

Specs are never comprehensive. In fact, until you have clear consistent acceptance tests, they are far from exact. A recent study indicated that the average requirements document is 15% complete and 7% accurate, and that the length of

time spent producing the document didn't significantly change this[3]. This is the “big lie” of BDUF. You simply can’t think of everything to the last detail and if the requirements aren't complete, the chance of the design being adequate is slim to none.

It is better to admit that the requirements are vague from the beginning, and work on the details in the form of acceptance tests as they are needed. That's really the hard part. It is seemingly impossibly in some cases until you have something half-working.. This is how it happens anyway, but usually the customer and developer expectations are all out of whack. Customers think they’ll get the implementation of the whole spec at once, but developers prioritize it and work on it sequentially. The developers will most likely have different priorities than the customer, so the wrong people are prioritizing. If you make customer prioritization the norm, both groups have the same expectation. We’ll identify all the big pieces (stories), we’ll prioritize them so that everybody knows what we’re working on and why, and we’ll flesh out the details when it comes time to do each one (tasks and estimates).

Our initial estimates are no more unreliable than the requirements. (Our estimates are at least 7% accurate). Once requirements are specified as unambiguous acceptance tests, we will get much better estimates. Until then (and beyond), we should work within reality.

Tests and Code are a better system documentation than anything else could be. It is also the most efficient way of getting accurate documentation. If you’re trying to learn a new part of the system, you eventually have to go from words to code anyway in order to really understand what’s going on. Why waste time on the overhead of keeping paper updated? We have never heard a good reason.

Earlier this year, Ken was in London and was a guest at the eXtreme Tuesday Club[4]. The first question he got was something like, "How does XP map requirements to code?". Ken's answer was "Why do you need to map it to code?". "So you can see how a the requirements are met." "The requirements are met when the acceptance test passes." "Yes, but what if the requirements changed". "Then the acceptance test changed". "Yes, but how do you know what change in requirements caused what change in the code." "Why do you need to know that?". "To make sure that the requirements are met." "The acceptance tests do that… Look, you are assuming that there is a requirement to track requirements to code. I'm trying to build a product that meets the requirements. Why would I waste time doing all of this mapping. The customer doesn't want a map of requirements. They want the

[3] Get Highsmith reference
[4] URL

requirements met. Of course they are going to change. So, we need a new test or set of tests and they have to pass." End of discussion.

XP gives you enough information to do your job as well as you can, without distracting you with too much.

Chapter 6

Having the Right Attitude

"Pride goes before destruction, and a haughty spirit before a fall."
-- Proverbs 16:18

> *No project succeeds without trust. You can't have trust without absolute honesty all the time. You can't have honesty without humility.*

You have to take a leap of faith to begin doing XP. You'll encounter resistance along the way, but now you're equipped to handle that. These challenges are nothing compared to maintaining the proper attitude while you're learning and doing XP.

XP will not work in an environment where bravado and spin are the norm. These things are the two biggest barriers to getting XP accepted and making it work. You need to nip them in the bud. The best place to start is by setting the example yourself. This takes more courage than many people can muster.

Honesty

Lying kills trust. Without trust you might as well not do XP. Customers and developers need to know without a shred of doubt that they are being told the truth all the time by everyone.

XP is not about diplomacy. It assumes (and demands) brutal honesty all the time. The practices make it harder to lie than to tell the truth. It forces transparency on a scale most people aren't used to. They often rebel against it.

XP forces developers and customers to be honest with one another. Take planning. Many people criticize XP for not planning ahead like many "heavy" approaches do. The problem is that those approaches are predicated on guesses about the future. That isn't telling the truth, no matter how good your guesses turn out to be. The unvarnished truth is that we don't really know what the future holds. Planning in XP is based on the principle that we admit what we know and don't know at every step. We put together a rough project plan based on rough estimates of rough requirements (stories). So does everyone else. If they don't tell you so, they are either lying or naïve.

We usually only put a detailed plan together only one iteration ahead because planning further is a fruitless exercise in reading tea leaves. Sure, you can try to work out more of the details based on the limited knowledge you have to look that far ahead, and you may actually get it close. Of course, the effort to do so will cost you some time now, and you'll use more time later either verifying your plans, revising your plans, or explaining why you aren't hitting your plans. Since we know we'll have to do at least one of these, we choose the more profitable. We revise or, more accurately, refine our plan based on the increased knowledge we do have when we get to that part of the rough plan. We assume that reality won't conform to the plan, so we keep it simple and easy to change. We keep iterations short so that we and customers can get frequent feedback on how things are going. That's honest.

XP also forces developers to be honest with themselves and with other developers. If you are writing code in pairs, at least one other person sees every boneheaded mistake you make. If a mistake happens to slip through (this *never* happens to us, of course), somebody will find it eventually because there is no code ownership. The preferred physical environment for XP has few or no walls, so people can see you struggling or goofing off. There are daily stand-up meetings where people get to hear (or ask) what you've been up to. If you want to be dishonest as an XP developer, feel free. But know in advance that it will be like denying you've taken cookies from someone else's cookie jar when they saw you do it and the crumbs are still in the corner of your lips.

XP doesn't reject tact. You should still respect the people with whom you are working. But XP demands brutal honesty all the time. The truth is going to have to be dealt with eventually. Ignoring it dooms you to failure, either in results or perception.

Humility

One of the guys at our company said on a recent XP project, "I love XP! I feel like an idiot every day! When I wasn't doing XP, I thought I was brilliant. Now I know I'm not, but I don't care because I've learned so much!" We thought about getting t-shirts made that say "Join the Extreme Programming Software Studio™…Feel like an idiot every day!" We haven't done it yet because we aren't sure how many others would see the splendor in the statement. Go figure.

You won't last long in a brutally honest environment without being humble. Other than quitting, it is the only reaction that makes sense. Your mistakes, weaknesses, selfishness and pride will be put on display. You can get humble real fast, quit because you hate embarrassment, or stay brazenly arrogant and be sacked by your team. This is a tough choice for many developers.

Developers are smart people in general. The ones known as “experts” will have the hardest time getting humble. Often, they’ll do whatever they can to avoid it. Ironically, the reputation they are guarding dwindles as they begin to be seen as arrogant. Work gets done without them. When they choose to participate, they’re behind everyone else. The rest of the team gets frustrated with having to catch them up all the time.

It isn’t any better for “experts” (or just good developers) who give humility the old college try. On just about any team we’re on, we find that we often have much more experience than others on the team. But not a day goes by when we don’t make mistakes that are caught by our pairs. When Ken coaches, he sometimes says apparently contradictory things. He can’t count the number of times he’s had to admit he actually was inconsistent. He has never regretted that. The result is always positive. Everybody learns. Sometimes, they learn that Ken can be an idiot, too.

Everybody on an XP team should feel like an idiot regularly. That’s healthy. Instead of paying lip service to the phrase “nobody’s perfect,” we realize it every day.

This whole humility thing may seem ludicrous to those who listen to us (and other Xpers) talk about how great XP is. But, please recognize the difference. We personally realize that we make mistakes and they are going to be found out. That gives us confidence that we are doing great work. Fewer mistakes go unnoticed. And, we are moving faster and more confidently than we ever have before. We may not be moving faster and more confidently than YOU ever have before, but we've seen the difference it has made for us. It is quite remarkable and you cannot deny that we have noticed a marked difference.

Sweet Freedom

We think disciplining ourselves to be honest and humble is worth doing for its own sake. But even if you’re not into character building like that, developing honesty and humility is great. They give you the freedom to maximize your potential.

Here is a stereotypical discussion between a developer and a customer in a "traditional environment":

CUSTOMER:	“I want everything yesterday, and I don’t care if it kills all of your people to get it done!” (we’re paraphrasing, of course)
DEVELOPER:	“Well, that’s a lot of stuff. Do you really need it *all*? Isn’t there some subset that’s more important that we could focus on first?”
CUSTOMER:	“It’s *all* critical. If we don’t get it all, my butt’s in a sling.”

DEVELOPER: "I hear you. We are in this together. We'll get it done. You can count on us."

A month later:

CUSTOMER: "You didn't deliver! I want this stuff *now*!!"

DEVELOPER: "There were some unexpected speed bumps. But I hear you loud and clear. We are in this together. We'll get it done. You can count on us."

Uh…this is rubbish. The customer is lying to the developer. Not everything is a number-one priority; that's being lazy. The developer is lying to the customer, too. He's committing to something that's ludicrous, and he's doing it with a straight face. When he fails (no surprise, really), he learns nothing and makes the same stupid promise again. The customer acts like he believes him. It's like a big game of charades.

What if it went like this instead?

CUSTOMER: "I know I can't have everything right now. I really don't need it. What I really need is Feature C. Everything else can wait, but we've got to have Feature C ASAP."

DEVELOPER: "Hmm. We have Feature B in this iteration already, and there isn't enough room for Feature C on top of it. Should be push Feature B off and do Feature C first?"

CUSTOMER: "Yep. That will give us what we need."

DEVELOPER: "Okay. Feature C it is. It's about the same estimate as Feature B, so it's an even trade. We'll get it done."

At the end of the iteration:

CUSTOMER: "Feature C works great! All the acceptance tests pass, so it's exactly what we were expecting. Can we get Feature B in the next iteration?"

DEVELOPER: "No problem. The next iteration used to have Feature C, Feature F, and Feature G, based on the original priority order. Feature C isn't there anymore, and it was the same size as Feature B, so we'll just do Feature B in the upcoming iteration. We'll get it done."

CUSTOMER: "I know. You have for the last four iterations."

That is honesty and humility in action. Which scenario would you prefer? We prefer to be free to tell the truth. If everybody expects this, and does it, everybody wins. If either side doesn't, you will crash and burn.

Come on! You say. Isn't that a bit contrived? Here's a real one from a few days ago:

We're not very far away from a release date, and people are under a lot of pressure:

CUSTOMER:	“I know you just got the relative date thing working by typing in a *plus* or *minus* followed by a number, but I'd really like to be able to specify *today* by typing in a *zero*.”
DEVELOPER:	“That's not going to happen. You said you wanted a *plus* or *minus*. That would be a new requirment and it would push the end date out.”
CUSTOMER:	(angrily) “Don't tell me that's what I wanted! I wanted a *zero* from the beginning and you talked me into the *plus* or *minus* because you said it would be easier to do.”
DEVELOPER:	“Calm down. I'm sorry I used my words poorly. It wasn't what you wanted it was what you agreed to.”
CUSTOMER	"OK. It's what I agreed to. Don't tell me it's what I wanted."
DEVELOPER	"Again. I'm sorry. I should have said it was what you agreed to."
CUSTOMER	"Well, how much longer would it take to make it work with a *zero*."
DEVELOPER	"Well, let's talk about it calmly. OK? (pause) I don't remember what I told you originally. The issue was that a *t* for *today* or a *y* for *yesterday* wouldn't work for internationalization. A *zero* would work theoretically, but it's ambiguous could be interpreted as the beginning of a valid date. As soon as you type a *plus* or a *minus*, there is no confusion and we can shift into relative mode. That date widget we are using from the 3rd party has some really raunchy code and it wasn't easy to get it to work with the plus and minus, but writing our own date widget at this point would be foolish. I don't think it will be difficult for users to get used to typing a *plus* or *minus* for *today* even though it isn't intuitively obvious. Of course, either is *plus* or *minus* for that matter. We're going to have to teach them to use *them*, so I don't see that it will be more than one sentence in the user manual to teach them how to use *plus-zero*, *minus-zero*, or just *plus* or *minus* to get *today*."
CUSTOMER	"Well, maybe not. But how much would it cost to make it work only when you type a single *zero* and *tab* out of there."

DEVELOPER	"Well, now that I've been in the code and pretty much figured it out, it might be a little easier. But, this isn't the same as keying off a *plus* or *minus* because when you type in a zero, you don't want to switch to relative mode. If you wanted a *t* or some other character that wasn't a number, that would be a piece of cake and I could do it for you right now. But I can't think of any other characters that would be more intuitively obvious that wouldn't cause problems for internationalization. There's also a lot of hairy testing to make sure it works for all the ways you can get to the point of one *zero* followed by a *tab*. I don't think it would take more than a day, but I'm not so sure there won't be some other gotchas in there."
CUSTOMER	"Hmmm… I still think I want the zero, but I'm not sure."
DEVELOPER	"Well, we'll just leave it out for now. If you are sure you want it, let us know and we'll put it back into the remaining list of things that still need to be done."

It wasn't perfectly smooth, but brutal honesty got us to the point of reality. Would "Yes Ma'am, the customer is always right" have gotten us a better result? Maybe it would have been better if the customer said, "Look here, it shouldn't take you more than a couple of minutes to make it work with the zero. So make it work!". If you think that's a better way to work, please don't apply to RoleModel Software for a job or to hire us to write your software.

It's not just about honesty and humility between a customer and developer. Think about interaction within a development team. Here is a stereotypical exchange at a development status meeting where bravado and deceit are rampant:

BOB:	“The Humphsplat class was a mess so I completely overhauled it to use different methods on DinkyDoo. It’s much cleaner now. I had to change the public interface a bit, but it shouldn’t be too much trouble, so I went ahead and migrated it to Integration Test.”
JANE:	“I’m depending on that class! You should have told me you were going to do that. Now my stuff won’t work!”
BOB:	(out loud if he's really brash, to himself if not) “You’re always saying that. You don’t want to do it right, so you leave junky code around. Just implement the changes and get on with it!”
JANE:	“I can’t get my current list of stuff done and upgrade to your new class. Unless you’d like to take on some of my tasks?”
BOB:	“No, thank you very much. I've got plenty to do already.”
FRANK:	“I’d love to help out, but I’m really having trouble understanding that part of the system. The logic seems really complex. Can either of you walk me through it? I'm sure it would help because I need to add logging to the DinkyDoo class.”

BOB: "The logic is complex because *some* people don't make it elegant like they should. I would walk you through it, but all my time is booked cleaning up *other* messes."

JANE: "I'd love to, Frank. But now, thanks to Mr. Senior Guru here, I've got too much work to do to take the time to explain it to you, much less get a good night's sleep."

And so on. Nobody is communicating with anyone. No problems, interpersonal or otherwise, have been solved. Everyone is defensive, caught off-guard, or continually ignorant without any hope of catching up. After the meeting, everyone returns to their cubes and gets deeper into the jungle without an escape plan.

What if it went like this instead? At the daily stand-up meeting on Tuesday:

BOB: "The Humphsplat class was a mess so Jane and I paired on that yesterday. We refactored it completely and it's much cleaner now. All the tests pass on the integration machine, so we're integrated and everyone else should be fine."

JANE: "That refactoring was tough, but I learned a ton. I understand how Humphsplat fits into the world now. Bob almost hammered the public interface, but I caught him and we worked through it."

Bob and I are done with that, so I'm available to pair."

BOB: "Hey! Well…yeah…I was going to…never mind. It was really stupid. I'm an idiot! But all the tests pass, so I guess I recovered gracefully. Thanks to Jane, that is."

JANE: "I better watch out or I'll get a big head. I wasn't that brilliant. Bob taught me a few tricks that I didn't know. Anyway, he and I are done with that stuff and I'm available to pair."

FRANK: "I'm really having trouble understanding that part of the system. The logic seems really complex. Can either of you walk me through it? I think it will help me with my next task. I have to add logging to the DinkyDoo class."

JANE: "Sure. I understand it now. I can walk you through that first and then we can pair on your next task."

Again, which scenario would you prefer? We prefer a supportive environment where everybody is learning, no one gets left behind, and nobody is a prima donna no matter how talented they are. If you have an environment like this, you feel like you can do anything. If you don't, prepare for battle.

When you're honest with others and with yourself, you have no choice but to be humble. When you are humble, you are teachable. When you are teachable, you

learn. When you learn, you can use what you learn to achieve great things. That’s winning.

Section Two: First Things First

We have now set the stage. Either you've already tried an XP experiment, or you have the tools you need to try one. Sooner or later, you'll take the plunge and start doing XP for real. When you do that, you have to know what to do first, why, and how.

In this section, we'll talk about what we think are the practices of XP that are essential to focus on first. These are the things that you must do, or everything else is moot. The other practices are important and will come into play. You can't get the full, synergistic experience of XP without them. But they aren't what you should start with.

For each of these essential practices, we'll specify why we think it's essential. We'll summarize how to go about doing it. Then we'll describe the best way to *start* doing it, since these things are probably the most foreign to you right now.

We'll also tell you some of the things we experienced along the way, or have heard from other pioneers that might help you avoid problems, or at least know how others have made it through the problems.

Chapter 7

The Bare Essentials

SOMETHING ABOUT FIRST THINGS FIRST OR TRAVELING LIGHT

--

> *Nail down the essential XP practices first. Without these, the others don't matter. If you get them, that will set the stage for implementing all of the others.*

Ernest Shackleton led the British Trans-Antarctic Expedition in 1914. At the time, he was one of the most experienced Antarctic explorers in the world. He prepared as well as could be expected. He took 27 men with him.

Within a year of leaving England, their ship *Endurance* was locked in the ice in the Weddell Sea north of Antarctica. The pack crushed *Endurance* slowly over a period of several months. Before it sank, the men salvaged as much material as they possibly could and took to the ice. Shackleton knew that they would have to spend an indefinite period of time living on the floes. There was only one thing to do.

Shackleton gathered everyone for a somber stand-up meeting. He told his men that they were going to walk out of there alive. To do that, they could carry only what they absolutely needed – the bare essentials. That meant only two pounds of "personal gear." When he finished speaking, he took out his gold cigarette case and several gold sovereigns, and without hesitation dropped them on the ice in the middle of the circle of men. They all followed suit. Coming out alive was more important than ultimately worthless baubles. The amazing ending to the story is that not a single man perished. They not only survived, they won.[1]

[1] This is quite possibly the greatest story of human survival and strength of will ever told. Check out the book *Endurance* by Alfred Lansing (ISBN 078670621X). If you don't, it's your loss.

Nothing brings what really matters into focus quite like staring death in the eye. Conducting a software project is no different. Don't believe it? Consider these sobering statistics from 8,380 software projects in a variety of industries in 1996[2]:

- 31.1% of projects were cancelled
- 52.7% were completed but were over budget, over time, or had fewer features than originally specified
- 52.7% of projects that broke their budgets cost 189% of their original estimates
- 16.2% were on time and under budget

That's as daunting as facing death on the Antarctic ice. Surviving and winning require that you follow a principle of XP. You have to travel light. When you begin the XP adventure, you can't carry worthless baubles with you.

So, the first thing you do is meet with your manager (this is true whether you are a manager or a developer… unless you are the big cheese, you have somebody you are working for). You say to them something like this:

"I've been examining the way I've been working and I think I could be doing a better job of helping you meet your goals. In order to do a better job for you, I need to be perfectly clear on what it is your goals are and what you want me to produce. I mean the big picture, not necessarily what you want me to produce this week (although I'd be happy to do that, too). Once I'm clear on that I want to examine everything I'm doing and make sure that I'm making the best use of my time in order to help you meet your goals."

After he picks his jaw up off the floor, he'll probably tell you something like:

"Well, we really need to get this product out (or at least to system test, or ready for the trade show, or…)".

He might also add something like:

"And, you know, I'm really under a lot of pressure from my boss about keeping Archibald in Marketing happy. So be sensitive to that."

We will almost guarantee you the first words out of their mouths won't be "attend as many meetings as possible and produce lots of documents that nobody reads." So, go ahead and ask them. Once they tell you what they want, hold them accountable to it. Whenever they ask you to do something that seems contrary to the

[2] Data are from Michael Mah of QSM Associates (as quoted in the *Adaptive Software Development* presentation given by Jim Highsmith at OOPSLA 2000) and a Standish Group International, Inc. study published in 1996 (available at \\Xp1\E\RMSProjects\website\ResearchMaterial\TSG -- Sample Research.htm).

goals, ask them whether that has become more important than meeting the originally stated goals. If they say yes, don't be a jerk, gladly do what they ask you to do. If it becomes a habit, find a good time to talk to them about it.

Nine times out of ten, when you ask your management what they want most from you and they see you sincerely trying to give it to them, they'll be thrilled. We've both managed people. Trust us. Starting here, two out of two managers surveyed want nothing more from their employees. You'll have to sample the other eight yourself. If you work for one that says otherwise, you might want to seriously consider looking for a new manager.

Now, all that's left to do is start doing the essential practices of XP on whatever part of the project you can and be sensitive to his other pressures (hopefully he didn't dump too many of them on you).

The XP Essentials

Arguments about whether or not a given project is actually “doing XP” aren’t productive. All of the practices reinforce each other, so they all are important. But we like to take a more practical stance here. As heretical as it might sound, there are certain things you must be doing before other things matter at all. That means certain XP practices are more important than others when you start. If you don’t have these essentials, the rest of the practices don’t matter because you’ll probably be hurting in a way that they won't heal:

1. Planning and estimating
2. Small releases (and iterations)
3. Testing First
4. Pair Programming
5. Refactoring
6. Continuous Integration

You may not get the rest of the project moving with you, but doing these things on even a subset of your project will get you going in a positive way and soon people will start noticing the difference.

If you are going to add any of the other XP practices when you get started, fine. We strongly suggest fasting from any other non-XP practices for at least a month, and being incredibly cheerful and friendly (which won't be hard if you are really doing nothing but XP).

Maybe you're saying, "surely, you can't mean every other practice". We still need to attend the XYZ meetings. And, of course, Sam needs to finish the design

document he has been writing since he is already 70% complete. We also have gotten pretty good at Microsoft Project for doing our project tracking, so there is no reason to change direction there. And our team is already divided quite nicely into the client group and the server group. And, Joe is already the "database king" and has that part of the system well under control. It wouldn't make sense to have him start writing tests or pairing with anyone. And…

Nope. We mean everything. Tell whoever you can that you are going to put everything but the essentials necessary to meet your boss's goals on hold for a while because of your tight schedule. (We're sure you have one… then there is a lot less risk in experimenting with XP because you won't be under a microscope). After a month, if missing something really hurt you, it is not like you left it 50 miles back on the ice and you can't get it back. But, how will you know how fast you can travel if you don't unload everything else.

If doing this will get you fired before the month is out, do just enough to not get fired. You will probably be so productive, that people will start noticing the productivity and start following your lead.

Remember that project we told you about earlier where Ken took over for another developer who had a family emergency? Well, when the emergency was over, Ken and the other developer, Joe (not his real name), overlapped a week in order to make a smooth transition as he handed back over the reins. On day one of his return, Joe spent most of the day with Ken to find out what he had done in the previous five weeks. By mid-day, Joe was quite impressed. "You sure have been busy while I've been gone." Two days later, Ken hadn't seen Joe other than in fleeting moments. They had arranged to have dinner together that night. Shortly into the evening, Ken started the conversation.

"Joe, where have you been the last few days?"

"You know, all those meetings."

"What meetings? You've just been back for a couple of days. What are you doing in meetings."

"Come to think of it, I've noticed that you don't seem to be in any of those meetings. How have you managed to avoid them."

"It's simple. Whenever Ralph (the manager, not his real name) asked me if I could attend a meeting, I'd say, 'Sure, I can attend if you think that's the best use of my time. You had previously asked me to get [something] done, but if you want me to push that out, it's your call.'. Most of the time, Ralph would say, 'Ya know, I'm not really sure if you'll be needed at that meeting. Tell you what. Could you be by the phone during the time of the meeting? That way, in case I think you're needed, I could just call you in.'. I'd say, 'I'll be right here, working on [something]. And I'll be happy to drop it when you call.' He just had never thought about the impact of

those meetings. Every once in a while, he'd say, 'Yeah, I hate to take you away from that, but I really think you need to be at this one.' So, I'd say, 'Sure thing. If you need me there, I'll be there.' I think I've attended three formal meetings since I've been here and I think Ralph likes the results."

"Are you kidding me. He loves you. You've made it hard for me to come back. I've got some pretty big shoes to fill."

"Nah. You can do it. Just learn how to say, 'I'll be there, if you think it's the best use of my time.' And make sure your getting good work done in between meetings."

Ken likes to drum a basic software development mantra into people's heads. "Make it run, make it right, make it fast." These core practices are what it takes to make XP "run" in your organization. Once you have these in place, you can make it right by adding the others to get the full, synergistic XP experience. Maybe you will even find a good reason to add something else you used to do (or some variation of it). Once all the practices are reinforcing each other, you can refine XP within your particular context. That's making it fast.

If you make it run first, you will be better equipped to overcome resistance that you are likely to face when you're trying to make it right and fast.

You can probably get away with "acting weird and being into that XP stuff" for about a month without getting too much resistance as long as you don't try force it on everyone else or break commitments to others without permission.

At the end of a month, you should have enough results to begin defending yourself if attacks come. But more likely, you will find others being interested in joining you and still others spending their time defending themselves…

Picture this. After asking your manager what he wants most out of you, you are successfully tackling it with XP and seeing good results. You have a great attitude toward your boss and your morale is going up and its starting to get contagious. Your boss has heard about some of the cool side-benefits of having the tests. You are moving faster than ever (partially because XP is helping you and partially because you are not doing a lot of other things that are less productive). One of your resistant colleagues seeks out your boss to complain. You figure it out… he's complaining that you're playing to win!

Chapter 8

The Rules of the Game

We need to refocus on our Core Values.
-- every CEO after a reorganization

> *Remember the four values of XP: Simplicity, Feedback, Communication, and Courage. These are the rules of the game when you're starting out.*

When you're starting out, there are a few principles to keep in mind that will make things much easier. These "rules" are based on the values of XP:

- Simplicity
- Feedback
- Communication
- Courage

Think Simply

Roy came to XP from a Big 5 consulting firm that will remain nameless to protect the guilty. He was used to a methodology that came on a CD…because that's the only medium it would fit on. On his first day working at Ken's company, Ken asked him to write a little code for a spike. Roy spent four hours on the problem and came up with something ridiculously complex, and he had a headache. Seeing his pain, Ken came over and "helped" him refactor. Within thirty minutes, Ken had an elegant solution with about one-third the code. Yes, Roy felt like an idiot.

It seems obvious that the simplest solution is probably right, but this has been forgotten. When you are taking your first steps with XP in the real world, nothing could be more important. XP is an answer to the question, "How little can we do and still create great software?".

Simplicity is easier than complexity in the long run. Certainly, coming up with the simplest thing that could possibly work takes some skill. Most often, though, the

barrier that keeps us from doing this is a predisposition to doing complicated things. Maybe this makes us feel smarter.

You should be like a child (we know saying this will be held against us forever). Kids often don't know the complicated way to do something. They just assume the simple way will work, and they go for it. XP requires you to do the same thing. Just do it and see if it works. This is especially true when you start.

Try the XP practices and see if they work. If they don't, respond just enough to make them work. Simple.

Get Feedback Early and Often

XP works only when you're getting lots of feedback all the time. You can't steer otherwise. Pay attention to the feedback you get, especially when you're starting. XP is based on some fundamental principles that don't change, but it has to be adapted to fit its environment. Changing it is not only acceptable, it's required.

When introducing XP at a client, we spent the first few days in a group working through some of the practices together. We'd start everyday by asking what concerned people the most; what they were most uncomfortable with. Before or after we broke for lunch, we'd ask if they were making progress in the areas they were uncomfortable. Before people left at the end of the day, we'd ask them again.

When Duff and Ken first did their three weeks of mini-XP, they started each day discussing what they were going to try to do to the best of their understanding of XP and what they were uncertain about. Then they'd have times of reflection several times during the day after that. They'd always end the day reflecting on what they had done well and what they could be doing better.

"I'm not sure how to write a test for this. It seems that the work is being done in the protected methods." "But I think we're better off just testing the public methods. Who cares if the protected methods change… as long as the public ones don't." "I'm not sure how to test that public method… maybe this is an exception to the rule. Maybe that's not even a good rule." "Tell you what. Let's assume it is a good rule and if we're still not OK with it in a half an hour, let's talk about another strategy."

"I think when we're pairing I'm not keeping up with you and then I'm not adding to the process." "Well, if you don't speak up, I'm not going to know when you've lost me." "If I spoke up everytime you've lost me, I don't think we'd get anything done." "Then I'm not doing something right. You are a bright guy. If I've lost you it's either because I'm not explaining myself well or because I'm doing something stupid and you are afraid to question me. I'll stop typing. You tell me what about what I've done you don't understand." "OK, then maybe after that, I should be the one typing. Then I'm sure I'll only go as fast as my understanding lets me."

"Well, do you think we accomplished a lot today or not? Are we going at a good pace?" "At times it felt pretty slow, but when I look at what we've got that we didn't have at the beginning of the day, I'm pretty amazed. And it's pretty cool that we have all those tests and we don't have to worry about that stuff breaking as we do more."

Just about anyone we've talked to who started XP did a lot of reflection on what they were doing for the first few days and/or weeks.

Listen to the feedback you get from your fellow developers, managers, and customers. Squeeze all the lessons out of that feedback that you can. Apply them as soon as possible.

Communicate

Success is directly proportional to communication. Talk with people about XP in your environment. There is no better way to learn. In *Planning Extreme Programming*, Beck and Fowler say that XP always changes within each environment in which it's applied. Without communication, these changes are impossible.

Most problems with projects can be traced back to at least one communication problem somewhere. Fortunately for geeks, you can't really do XP without communicating.

XP forces all parties to communicate by employing practices that can't be done without it, such as pair programming. If you don't at least communicate with your pair, you'll have to be mute all day, which is hard even for geeks. And your stand-up meetings will be really weird.

Be Brave

You had the guts to take that leap of faith and give XP a try. Have the guts to see it through. Don't give up when things get tough. That is too easy. Remember Brave Sir Robin in *Monthy Python and the Holy Grail*. "When danger reared its ugly head, Sir Robin turned his tail and fled." Don't be like that. Play to win.

Chapter 9

Exception Handling

"Plurality should not be posited without necessity."
-- William of Ockham

> *Assume the practices of XP will work. Don't come up with elaborate schemes for handling exceptions before you encounter them. When they come up, handle them as simply as possible.*

People who have difficulty believing XP will work often list off a bunch of exceptions:

- How do you pair if you have an odd number of developers?
- What if the customer won't write stories?
- What if the customer won't write acceptance tests?
- What if management refuses to set realistic delivery schedules?
- What if management doesn't believe your estimates?
- What if management refuses to let you pair program?
- What if the cost of tea in China doubles?

How do you handle these proposed exceptions without just ignoring them? Ignore them until they can't be ignored. Then handle them as simply as possible. That is the simplest thing that could possibly work.

Handling XP Exceptions Like Code Exceptions

Writing code is easier if you can count on the methods you are using not to throw exceptions. You can just invoke the method and expect a certain result. This lets you proceed with confidence. In fact, we have found that programming goes better when we assume exceptions don't exist. We find out soon enough if we're wrong. Then we handle the exception with the simplest approach that could possibly work. That almost always works.

If we went to the other extreme, we would be afraid to write any code until we knew how to handle all of the possible exceptions, and all of the exceptions in our exception handlers. This is paranoia. That way lies madness.

Implementing XP in an organization that's not used to it is the same. Assume there aren't any exceptions to handle. The practices of XP are simple, although not always easy. It has been called a "lightweight methodology" because none of the practices requires a lot of ceremony or training. Each is simple to implement and gives the results it advertises. Assume they work.

When you find an exception, handle it as simply as possible without turning it into something completely different that misses the point. Don't come up with an elaborate scheme of exception handling that turns XP in a heavyweight methodology. Just handle the exception and figure out the best way to keep an exception, rather than a rule.

Building Sidewalks (THIS DOESN'T REALLY FIT HERE, SHOULD BE EARLIER IN THE BOOK WHEN WE TALK ABOUT BDUF – XP IS LIKE BUILDING SIDEWALKS!)

Roy's father has a theory about building sidewalks (he's a little odd this way). He says that you shouldn't build *any* until you know where they should go. Construct your building first. Let people walk around among them for a couple months. Then build sidewalks where people have worn paths in the grass, no matter how unorthodox your sidewalk pattern looks. Those sidewalks will be used.

Do the simplest thing that could possibly work. Start with the essentials and do XP as prescribed. When you can't, handle the exceptions as simply as possible.

Here are just a few exception handling routines we've used:

The OddNumberOfDevelopers Exception

On one of the first "pair programming" days at our first big XP client, we had an odd number of people. Well, if all production code has to have two sets of eyes on them, the solution is NOT to let someone write code by themselves. So, we tried a couple of things. We tri-programmed for awhile. It was OK for a while. To some of the newbies, every line of code was new and exciting. But after a while, that got old. So, Ken let the two newbies continue on their own, and he went to check on the other pair. They were doing fine, so Ken went and read the manual which described the protocol that was going to come in the serial port.

(You don't need to pair-read, although we have found times that two people reading similar things in parallel provide some benefits).

When Ken had a clue about where to start on the SerialPort task, it was almost 3:00, and one of the developers had to go pick up her daughter at her day care center. Imagine that, one pair stopped what they were doing and another one formed. Cool.

There have been times when the hard part of a problem is solved, and both developers in the pair know exactly what's going on. One party gets interrupted by someone else or by nature. The other developer might take the next few steps without him. When the partner gets back, they are shown what's happened. If they went too far, they have to undo it. If not, move on. Excuse us for being practical.

Does this ever backfire? Absolutely.

One day, Ken and Duff were pairing together. During the day, their pair was interrupted four times. It went something like this.

Ken was pulled away for a couple of minutes. A few minutes later, he came back and took the keyboard. Within moments, he was confused. A method he was trying to call on wasn't there. He knew it was supposed to be there, he had just written it earlier that day. He was speechless. Duff said, "what are you looking for?". Ken said, there was a method called something like [xyz] that I was counting on. Duff said, "I deleted it.". "Why?". "Nobody was calling on it except the test, so it wasn't necessary." "Yes, it was. I put it there yesterday as the first part of this task because I knew I needed it in order to make this task work." "OOPS. Man, you leave me alone for two minutes, and we've just lost ten mimutes."

An hour later or so, Duff went off to relieve his bladder and Ken browsed a couple of methods looking for one that they might be able to use when he was rejoined by Duff. During his browsing, he noticed some superfluous code in one of the methods, so he changed it. Duff returned and they finished their task. All the tests in the related test suite passed. Ken suggested they integrate. Duff said, "before we do, let's run this other test suite which is kind of related, just in case we broke anything." Ken said, "Nah, we didn't touch anything there that would have affected it." Duff said, "Well, just humor me." ARRGGGHHH! A red bar! Duff incredulously states, "The [blah] test. How could that have failed?". Ken sheepishly responded, "I think I know…".

Similar scenarios happened two more times that day. We went four for four. It was quite comical. Have we stopped doing little snippets of code when we are interrupted? No. But we are a little more mindful of showing each other what we did while the other was away.

Most of the time when you have an odd number of people, there is something that can safely be done by one person. When that person is ready to roll, it is often possible that another person can find something safe to do as an individual. If not, think of something. If not, tri-program for a little while.

As long as pairing is the rule, and exceptions are few and far between, you'll be fine. In fact, the problems that come up when you make too many exceptions will become obvious, and you'll discover your own limits as long as you are honest and humble.

Get over it. Play to win!

The CustomerWontWriteStories Exception

The customer at one of our client's site has a tough time, for some reason, picking up a pencil or keyboard and writing stories. However, they don't have any problem telling us what they want. So, they tell us what they want. We write down what we think we heard and ask them to verify it. (In many ways, that works better, because we know we understand what we wrote). This is nothing to panic about. We need user stories. They won't write them down. The simplest thing that could possibly work is that we write them down based on their input.

If you can get the user to verify the stories, you end up with roughly the same results. The point is that the customer is still the one identifying the business requirement and setting the priorities. They just had an assistant which happened to be you.

Get over it. Play to win!

The CustomerWontWriteAcceptanceTests Exception

See the CustomerWontWriteStories exception.

If you can get the user to write the acceptance tests, do your best to write them and get them to verify them. By verifying them, they are signing off on them. If they refuse to verify them and won't offer you an alternative, they've indirectly verified them. Eventually the truth will come out and somebody will try to fix it. If not… Hey, at least you've got some set of tests you can run to verify you are actually building something. It's better than you had before XP.

Get over it. Play to win!

The ManagementSetsUnrealisticSchedules Exception

Tell them that you don't think its possible, and give them a date/scope that is possible. In the meantime, tell them you'll do what you can reasonably do to meet their schedule. The more realistic schedule will come closer to matching reality.

The big concern here is that they blame XP for missing the schedule. One strategy is to ask them what they expect to get done in the next month. Estimate how long you think it will take to get those items done by breaking them into small chunks, just as you'd do with XP. Do what you can their way, and size how much they are off by. Then, point out the discrepancy and ask if you can try a different approach for the next month.

As long as they offer unrealistic schedules, reality will keep hitting them in the face. Eventually they'll either

a) get the hang of it and stop setting schedules that way, or

b) they'll lose their job, or

c) you'll lose your job, or

d) nothing will happen.

The only bummer out of this is C. And you can look at that as a blessing or at least as something you might not have avoided if you never tried XP.

On A, B, or D you can run the project according to the more realistic schedule you set. A is certainly the best situation to be in. On B, you might get new management that is just as bad, but you might not, and at least you have a fresh start. On D, you are probably in one of those organizations that have no accountability. Personally, I'd look for another one because it shows that people either aren't honest or aren't competent. Another alternative might be to take advantage of it and do the job well (i.e. play to win!) and stand out as some of the few people who actually do what they say they are going to do. Maybe others will follow. At least you can sleep well at night in your integrity.

Get over it. Play to win!

The ManagementDoesntLikeMyEstimates Exception

Ask them if they know something you don't that will help you get it done quicker? If they do, great. If not, tell them you are sticking by them and tracking them. You would be delighted to find out that you were too conservative.

Otherwise, see the ManagementSetsUnrealisticSchedules exception.

The MyManagementWontLetMePair Exception

This is one of the toughest ones to deal with and it is really hard to do this and be honest. However, if you can find another word that doesn't trigger the corporate gag reflex, use it.

Dan Rawsthorne conveyed how he did a lot of "mentoring" because "pair programming" was not acceptable in the "high ceremony, DOD" project he was on… Mentoring good. Pair Programming bad.

You'll have to be a little more creative to explain who is mentoring whom at times, but try it.

The CostOfTeaInChinaDoubles Exception

Get over it. Play to win!

Chapter 10

Communication

asdf.
--

> *If you aren't communicating well, XP won't work, no matter how many of the practices you try to implement.*

We will get to the essential practices, we promise. But before we dive in, we need to spend some time on an XP value that can get lost in all the talk about the mechanics of XP.

We told you in the last chapter that success on any software development project is directly proportional to communication. One of the brilliant things about XP is that it forces developers to communicate in order to get anything done. If left to their own devices, most developers wouldn't talk much. XP makes that more unnatural than not communicating.

Think about the person with you have the most intimate communication. Do you ever produce a document for them to read so they know what you've been up to? How about a weekly status report? Do you drop them an e-mail when they are in the same building as you? When you're concerned about how they're doing, do you schedule a conference room, invite a lot of other concerned people to come, and then intimidate them into committing to doing more in less time?

We hope not. People in the same room talking to one another have the highest bandwidth of direct and indirect communication. "Ken's got that look again…what am I doing wrong?" "Andy, you look frustrated…is it with me?…do I need to slow down?" "We're stuck…can anybody help?" Somehow, geeks and their managers think that they can improve on that communication by using formal processes or technology. Give us a break.

XP using four things to force and facilitate face-to-face communication among the people involved in developing software:

1. pair programming (and switching often)
2. stand-up meetings every morning
3. planning (essentially, talking to the customer and to each other a lot)

4. a team atmosphere and environment that encourages impromptu communication as the first line of defense

Pair Programming

We'll talk about more detail on the mechanics of pairing later (Chapter 13). What's important to note here is that pairing makes programming an exercise in constant communication, both within and among pairs. It does this in two ways:

1. replacing most written documentation with oral history and explanation
2. disseminating information through the "pairvine"

The theory goes, if you have everything documented then losing people is less of a risk, because new people can come up to speed more quickly. It also should keep everyone on the project in synch with what's going on, thereby maintaining design consistency.

That doesn't work. On every "well-documented" project we've ever seen, the pace of production was lethargic, it took forever to fix a problem (unless you broke the rules), and the design was notoriously complex and confusing. New people don't come up to speed quickly because the documents are outdated or incomplete. If they want to find out what those documents mean, they talk to people anyway.

It seems that one of the curses of more heavyweight processes is that they treat people as commodities. Not surprisingly, they usually end up with people who blindly follow a process and aren't allowed to think effectively. It looks very much like the "nameless horde" in that famous "1984" commercial Apple used to debut the Macintosh. It strikes us as asinine to have a small group use their brains to document an approach to the problems they think they have, and then have other groups use their brains to interpret what the first group wrote. Maybe they'll have a few cycles left over for programming.

We would rather encourage people to use their brains and collaborate to solve real problems. Pairing does that by having people spend their time in the code. When new people join, they pair with folks to learn rather than being handed a document when they walk in the door. Pairs rotate frequently. That means code knowledge gets passed around, as do development lessons, tricks, and so on.

A good illustration of the principle comes from an ongoing project Ken is a part of. As he tells the story

> *I've used VisualAge for Java for quite a while. Wherever I go, I'm pretty much considered the "VisualAge guru." But you know, there are features I still don't know. One day, Duff asked me a question about*

VisualAge I couldn't answer, so he went off to study the online help a little bit. While he was doing that, he picked up a great trick for using CTRL-Space to fill in method names automatically while you're typing. I was out the next day. The following day, I was pairing with Karen. I had the keyboard, and was just about to type out a long method name. Karen said, "Stop, let me show you something." She hit CTRL-Space. The least experienced programmer on the team taught me how to use this powerful feature…The next day, I was pairing with Andy and he asked, "What's the name of the method I need here?" Ready to show off my new trick, I took the keyboard and hit CTRL-Space. Andy said, "Oh yes, of course." Surprised, I said, "You already knew about CTRL-Space?" He said, "Sure, I think it was John who showed me yesterday." Within three days, everyone on the team had learned the new trick, including a person who would have missed the announcement had it been made. No one had to put together an e-mail describing how to use the new feature. No classroom required.

If you are switching pairs around frequently, most of the verbal communication you'll need to avoid surprises and to disseminate knowledge will happen. We're not making that up. Now, for the real kicker. If you add Stand-Up Meetings, we predict that more than ninety percent of that communication (some would say one-hundred percent, if things are really clicking) will happen.

Stand-Up Meetings

Nobody will know everything all the time. That means everybody will require some outside knowledge at certain points. Pairing is the first step, but what if you don't know the details of what other pairs are doing, how can you know what they need to know? And how can you get knowledge that you don't have when you need it? The answer is the Stand-Up Meeting.

Trying to plan who will need to know what and when is a fool's errand. You don't know everything yourself right now, and you don't know what you'll need know tomorrow, because the problem will be different by then. Everybody else is in the same boat. The trick is to communicate a little about what everyone knows, and find out what they don't, on a regular basis. That way, people on the team can identify who has the knowledge they don't have. That is what a stand-up meeting is all about. It is a chance to hook up the people who have knowledge with the people who need it. If you do it on a daily basis, the chance of missing the opportunity at this important rendezvous are greatly reduced.

In fifteen minutes (or less, depending on the size of the team) you can:

- get a sense of the trouble spots
- identify who might be able to help

- communication surprises to exploit or prepare for
- make sure you are starting the day right

Replace your regular meetings with Stand-Up Meetings. Get everybody together in one place and stand in a circle. Go around the circle and have each person share what he did yesterday that might affect others, what progress he made yesterday, and what he plans to do today. The only discussion allowed is the asking of questions that have simple answers. Any longer discussion should be taken off-line. It's as simple as that.

We do Stand-Up Meetings every day on all of our projects. It's a habit. And it's amazing what gets done. Laurie Williams from North Carolina State University stopped by late last year to bring a few of her students to be "flies on the wall" for a day. She stuck around for our Stand-Up Meeting, but had to leave early for another appointment. She interrupted Ken just long enough to say,

> *I hate that I have to leave. This has been amazing. The number of issues that were raised and addressed in such a short period of time in your Stand-Up Meeting is phenomenal.*

And Ken thought we were having an off-day.

If it's so obvious that you should have Stand-Up Meetings, why don't people do it all the time? There are several reasons:

- you may have problems getting a quorum
- people with the most knowledge might not share in the meeting
- people with the most knowledge might try to share it all in the meeting
- people might go into elaborate detail
- it might be difficult to find a place to stand up together

If you have trouble getting a critical mass to show up, examine whether you've communicated the reason for the meeting. If you haven't, do it. If they still are skeptical, ask them to humor you with a one-week experiment. They will be addicted in short order.

If people with knowledge seem reticent, identify why they won't open up. Many times, the ones who think they'll get the least out of the meeting will be the ones most likely to resist. They might feel threatened, because they feel their knowledge gives them status, or they are scared it will be revealed that they don't know as much as others think. Try encouraging them by acknowledging their importance and asking them to share. If that doesn't work, ask your manager to encourage them.

Stand-Up Meetings are supposed to be short. If people spout off, Stand-Up Meetings get long. Appreciate their knowledge, but remind them that the meetings are supposed to be short. Direct other people to them for more information, but tell them to take detailed discussions off-line. Thank them in advance for being available for off-line discussions. Ask people to answer only the three questions:

1. What did you do yesterday that might affect others?
2. What progress did you make yesterday?
3. What do you plan to do today?

To develop the habit of being brief, interrupt long-winded speakers, but then yield them some extra time when you suspect that there are a significant number of people who want to know more. These folks are typically searching for significance. Once you've given them the floor, they'll be less prone to take it unnecessarily.

If you can't find a place to stand up, ask management to rearrange your space. It's not a lot to ask. Stand-Up Meetings take up much less space than is available in a conference room. If rearranging the space simply can't (or won't) be accommodated, ask someone in management if you can use their office while they're trying to find a more convenient place. Make sure they understand that scheduling a meeting room daily for a fifteen-minute meeting blocks other people from using that space. It's also wasteful to make people travel five or ten minutes to get there. As a last resort, you could try meeting in a hallway. It's not ideal, but it usually keeps people from rambling, since they don't want to disturb others. It also sends the message that it's more important to communicate than to travel to meet or to be comfortable.

We would go so far as to claim that the Stand-Up Meeting ought to be the thirteenth XP practice. We're certainly not advocating an explosion of practices. The existing twelve are great. But we've talked to so many people who claim to be having trouble implementing XP who aren't doing Stand-Up Meetings. We think they wouldn't be having so many difficulties if they were doing them. We also think that the need for Stand-Up Meetings is not as explicit as it should be in the XP literature, or more people would be doing them as essential part of the discipline. Discuss.

Planning

Planning in XP is very different from they way it's done in other methodologies. This is a good thing.

Heavyweight methodologies tend to replace communication with formality and documentation. This is most obvious with regard to planning. Various groups on the

project develop their own bottom-up estimates of work based on their understanding of the requirements and of how they relate to the rest of the world. They develop complex lists of dependencies and milestones. They have "checkpoint" meetings. They disseminate status reports. They have contingency plans in case something goes wrong. And they still get screwed up.

Roy worked on a huge project at a nationwide bank in the U.S. where all the managers did was plan. At one point, between Release 1 and Release 2, the management team spent over two solid months planning. The plans were so complex, you needed degrees in finance and logistics just to read them. It seemed like all the possible bases were covered (they better have been with all that detail). But the project struggled constantly to keep its head above water. The plan was effectively useless for keeping everyone informed and maintaining control.

The reason these other approaches fail is simple. There isn't much communication going on. There is a lot of talking, and certainly a lot of paper, but there isn't much communication. They say, "Make the plan, then follow it religiously. All will be well." But there's something rotten in Denmark. Requirements change. Your bottom-up estimates that you worked so hard to make absolutely right…will be wrong. The complex dependency charts will miss something.

The only solution is to increase the bandwidth of communication. Planning in XP requires constant communication. Documentation is minimal (some note cards are about all). Customers have to talk to developers, and vice versa. Everybody has to listen. If they don't talk and listen, they will have no clue what to work on next.

Atmosphere And Environment

XP requires an open, group workspace to be effective. Why? Because Stand-Up Meetings, planning, and pairing still aren't enough communication. Perhaps you're getting the point that communication is paramount. We hope so.

Stand-Up Meetings and pairing cover about ninety percent of the communication that has to happen to keep the team humming. It's the remaining ten percent that we're talking about here. Often, people run into trouble in the middle of the day. If it is their nature not to interrupt others, or to avoid asking for help, they'll probably wait until the next morning's standup. That's suicide. Nothing will kill your velocity faster than this (except a freak asteroid accident, but that's another book).

People on your team need to start forming the habit of looking around at their teammates whenever you have a break in the action. If they see frustration, concern, anxiety, strange behavior, or simply lots of silence and little typing, they need to ask those folks if they need any help. They can try to get them over the hump, or ask

them for help on something to get them away from their problem for a while. That could break the logjam.

When emergencies happen, everybody needs to be within earshot. Someone accidentally deletes something that everyone is using (hey, it happens). He says, in a louder-than-normal voice, “TEAM, we’ve got a problem. I just deleted the project files by accident and I need help recovering.” (By the way, this is not a contrived quote). Or maybe somebody at the integration machine can’t integrate their stuff because the existing tests don’t run. “TEAM, we can’t integrate. We’re getting a resource error. Can the last person over here help us out?”

Everybody on the team should be behaving this way. Looking around. Helping out. Keeping people out of ditches. Asking for help when *they* are the ones in the ditch. Letting people know vocally about emergencies that might affect them. You can’t do that with walls in the way. An open workspace where everybody can see and eavesdrop on everybody else is critical.

[we need to put a picture of our studio here… we should probably put other pictures in the book, too].

[we might want to put in the floor plan, too. The floor plan here. The floor plan at OTC… anywhere else doing XP].

If you have to live in cubicle land, you can still do it. If the people you are working with aren't close by, make sure people know that this is keeping you from being as productive as possible. If it can be done, remove and/or rearrange the cubicle walls… they are supposed to be moveable. Below is an example of some things we've seen work pretty well.

[need to add picture here]

But don't let lousy cubicles be an excuse. In the early days of our first bix XP client, we had to do all of our work in cubicles that were barely big enough for one person. The only place the monitor would fit was in the corner. The keyboard was on a shelf under it. To reach the mouse or trackball, you had to reach under the desk, and your hand had about an inch of clearance. We did that for the first three months of the project. It was awful compared to anywhere we've worked since. But the client was convinced it was the way to go.

They extended our contract and found a small room that had been abandoned and let us move up there. The project manager found some old desks that were being scrapped that didn't have any drawers attached to them so people both fit there legs under them. We arranged the room like this at first because we didn't think we'd be able to arrange them in a way where we could face each other:

[picture needed]

It kind of worked, but people would bump into each other in the corners.

Then, after a few months, someone had an idea for rearranging them. They brought in a tape measure and figured out we could do it. We rearranged the furniture like this:

It made rolling chairs from place to place a bit difficult, but people could see everyone else without having to turn completely around and it improved communication and lessened the amount of chair collisions when people weren't intentionally moving them.

Our studio is extremely spacious compared to this, but it works.

And don't forget the whiteboards!

Alistair Cockburn has pointed out that the most efficient form of communication is two people at a whiteboard[1]. Good ones can be expensive and you might have problems getting people to do the capital outlay. In our studio, most of the walls are covered with whiteboards (and the whiteboards are usually covered with all sorts of remnants of discussions). We priced good whiteboards and felt there had to be a cheaper way. We went down to Home Depot and bought Mylar paneling for about $20/sheet. Each sheet is 4' x 8'. We bought 10 of them and a bunch of tubes of liquid nails. Some parts of the wall could not accommodate a 4' x 8' whiteboard. Nothing a saber saw with a fine blade couldn't handle. So, we have whiteboards everywhere you turn and it cost us less than $300 plus a bit of manual labor.

At the XP client who had small cubicles it was hard to find a place to hang a whiteboard, so we bought some static cling white board sheets and put them on the sides of the cubicles. Almost none of the conference rooms had whiteboards, either. I've never seen so few whiteboards in a corporate environment. Even when there was a whiteboard in a conference room, it was usually very small, and it was hard to find dry-erase markers.

When we moved to the abandoned room, getting it outfitted with whiteboards was not in the budget. So, one of the people in the group went to Home Depot and bought 4 Mylar boards for $80 and submitted it as an expense. In a week or so, "facilities" showed up to screw them into the wall. We probably had more square feet of whiteboard in that room 300 square foot room than were present in the rest of the 300,000 square foot facility! [need to check on the size of the actual facility]. I know we had better communication.

Communication doesn't just take place among the development team. You can improved your development process immensely purely by getting your developers communicating to each other via the means we've stated above and others.

[1] Get the reference from this… one of Alistair's papers.

But that is only part of the equation. Communication is vital between business and development if you are going to play to win.

Read on.

Chapter 11

Planning and Estimating

Life is what happens while you're busy making other plans.
-- John Lennon

> *We won't be right, but we can avoid foolishness with a few simple tools. We can get better by refining things. To do that, we have to have something to refine. So we plan and estimate.*

[There's way too much in this chapter… we need to break it down and reorganize]

Planning the XP way formalizes and reinforces the separation between business decisions and technical decisions. (Who says XP is totally informal?). It lets customers and developers learn their roles and teaches them how to talk to one another. This is essential to your project success.

Setting a course and steering is the way we plan XP projects. In fact, XP projects are one long drive. Steering keeps us from crashing in the present due to our bad guesses in the past.

Perhaps most important, communicating about planning and estimating is the primary means of helping customers and developers to embrace change without fear.

Learning Roles

At OOPSLA 2000, someone asked Kent Beck to boil XP down to its essence. The first thing out of his mouth was "Separating business and technical decisions." We agree. Planning is the first step in that direction. It is where developers and customers learn and practice their roles.

Alistair points out that two people at a whiteboard is the most effective means of communication. We would suggest that a business person and a technical person manipulating story cards on a table is a close second.

Customers write stories (or even just a good name for the story) on cards and prioritize them in the Planning Game. This gives them continual practice in specifying what the system is supposed to do and in prioritizing business value for developers. When developers ask for clarification on something, the customer gets

practice in refining stories. These exercises force customers to learn how to talk to developers honestly as partners.

Developers break stories into tasks and estimate them in the planning game. This gives them continual practice in understanding what customers are saying. It helps them learn how to estimate as accurately as possible, based on past experience. It helps them get used to probing for the customer's meaning underneath the words on index cards. When in doubt, ask the customer. These exercises force developers to learn how to talk to customers honestly as partners.

These roles are one of the important points at which XP lays down the law. Customers decide what gets done, and in what order. Developers decide how long it will take. Unless all of the players respect these roles (and play their own), XP won't work. Prepare for lots of problems down the road.

Developers cannot assume a given feature is necessary unless the customer tells them so. A developer can't make stuff up. If the customer doesn't put it on a card and prioritize it, it doesn't exist. If the developers don't understand a requirement, they ask the customer. If they have more time than they thought, they ask the customer what to work on next. If they have less time than they thought, they ask the customer what to defer. When in doubt, ask the customer. Then follow instructions. As Kent Beck says, "Requirements is a dialog, not a document."

Customers must accept developer estimates without question, trusting that the estimation process will be self-policing over time. When the developers tell the customer that something has to fall off the plate in order to satisfy the customer-established priority of stories, the customer has to tell them what falls off. There can't be any bickering. You'll get it next iteration if it's the first priority of what's left, end of story. No fair "holding their feet to the fire" to get more in less time. That's against the rules. It's also against the rules of reality.

An Introduction To Reality

Everybody needs at least one CASE tool. No exceptions. If you don't have a CASE tool, you are doomed to failure. You are living in a fantasy world. And, if you buy your CASE tool from a software vendor, you paid too much. Don't buy the training either. All the training you'll need is write here in this chapter.

The best CASE tool is found at office supply stores everywhere and it comes in a variety of sizes, colors, and designs. If you don't currently have at least one variety at your disposal, chances are good you can get your hands on it in short order without having to fill out a purchase requisition form. Oh, how foolish of us, we broke the rule and used an acronym without defining it first…

Cardboard Assisted Software Engineering (CASE)[1] is done with a stack of index cards. The resulting product may not look as polished as those produced by your favorite software-backed tool that produces pretty diagrams, but the positive effect it can have on your project is quite remarkable. Index cards, when used correctly by a trained professional, have the power to make both business and technical people face reality!

(NOTE: It does not have the power to make them like "reality" or deal with it rationally, it just makes them face it).

Let us share several stories that we've dealt with over the last few years.

I'd Really Like to Help

Remember how Ken shared that he really messed up when he introduced XP to his first client. Well, that is only partially true. He messed up in that he wasn't as gentle and patient as he should have been and sometimes was focused on the wrong thing. It was at the same client that Ken began to see the power of XP to chart a course to the land of reality.

The client was under a lot of pressure to raise his next round of capital from Venture Capitalists. They only had 2-3 months worth of money remaining. It seemed pretty obvious to Ken and the founder that the Framework he was working on for them was the foundation to their long-term future.

In the first few weeks of Ken's engagement at the company, his task was to learn the prototyped framework and some of the technologies it was exploiting as well as make recommendations for where to go with it. After doing so and spending a bunch of time in discussion with the founder, it was clear that the framework needed to be tightened up. The client agreed to this and the direction had been somewhat clearly set before they left for a three week trip to California for a series of shows, sales calls, and fund-raising efforts. It was agreed that Ken would work with Duff while they were gone in "XP fashion".

Those three weeks were considered "Iteration One" even though the client didn't call it that. We had to create our own stories based on what the client had said verbally before they left. We've already shared what great progress was made and it was clear that the footings for the foundation were laid. Not only had we accomplished a tremendous amount in those three weeks, but we had gotten quite good (relative to where we started) at XP. We had broken down the stories into very small tasks and estimated them before we started them. We paid attention to how well we estimated and were getting pretty good. And we had a lot of candidate

[1] Thanks to ??? at the Extreme Tuesday Club for giving us this one.

stories to share with them that we felt would be needed to finish the foundation in the next few iterations.

While the client was in California, we had occasional conversations. They told us each to work on a couple of different things until they could go over the details of what we had done with them. Duff and Ken were not convinced that they needed to Pair program all the time, so they went their separate ways and tried to communicate with each other several times a day. (They were both working from their houses at the time). After a few days of working by themselves, even with frequent communication, they were both getting a sense that they were working more slowly and that the quality of what they had produced had gone significantly down… Duff's significantly moreso than Ken's (who was more seasoned), but both were noticeable.

The client came back from California and were "ready to meet" a few days later. By this time, Duff and Ken were ready to propose doing XP exclusively until proven why they shouldn't. The client was thrilled with what we produced, but felt we'd be better off working on our tasks by ourselves. "We can't afford to pay two developers to do one thing. It's just not realistic." (see Pair Programming section).

Ken and Duff both accepted this for a while, but as they saw both of their quality going down and their work getting harder to sync up they started pushing harder to work together and do XP. They also wanted the founders to join them, telling them how much they'd learn by doing it. The client's said that they didn't have time, so they asked Ken to produce a document describing the framework, while Duff was supposed to work on a Configuration application for the framework.

Ken and Duff's attitude got worse, and the clients started to try to build stuff on top of the framework. Unfortunately they were building there stuff late at night and in the wee hours of the morning. (They were too busy with business things during the day). Occasionally Ken and Duff started getting e-mails that would say something like, "I was having problems using the framework last night. Finally, at around 2 AM, I figured out how I could use the Bus to X, but I'm not sure how to make it do Y."

We thought we had documented it well, so we pointed them to that point of the document and said that if their questions weren't answered, we should just sit down together and work it through. They "didn't have time to sit down together… but thanks for the pointers". Eventually their frustration with the framework was getting higher but communication was getting worse. We suggested pairing with them and they started getting upset. "No, we told you we can't afford to have two people working on one thing."

In addition to this, most communication came in the form of directives or suggestions. "Could you add this feature from the prototype back in.". "I need the framework to do that". "It would be good if you could…". "Flush out the

documentation more." When Ken asked for help in understanding what they were asking for, it was difficult to get an answer, but he'd often get a couple of more directives. He was also told to watch his hours, because money was getting tight until they could secure the next round of funding.

It was pretty frustrating. He wanted to help but felt unable to. So, he came up with a plan. He put everything he was asked to do on index cards in the smallest chunks he could imagine and put an estimate of number of the number of days he thought it would take to complete each. Most of the numbers ranged from 0.5 days to 2.0 days. The ones he wasn't clear on had significant question marks next to the number, usually meaning that he needed more information about exactly what they wanted because there was some inferred ambiguity. He had between 40 and 50 cards. Duff started making his own set of cards. At times it wasn't clear whether Ken or Duff were supposed to be working on a task because they were both asked similar things in different conversations.

Right around the same time, the founders said that they could only commit to about 200 more hours of work. So Ken took advantage of the opportunity. "I appreciate your limited funds and would like to sure you get the best out of these 200 hours. I've been writing down all of the things you've asked me to do and it adds up to a lot more than that. Can we get together sometime in the next couple of days for an hour or two and prioritize the tasks? That way we'll both know exactly what we are expecting and aren't expecting."

The meeting was arranged. It started out pretty amiable. Ken explained that each card just had a sentence or two about each thing he had been asked to do, and explained the numbers. He further explained that those numbers didn't include any time for communication with them or any other "overhead" so that they might want to multiply by a fudge factor of 2 or so to figure out how much he could actually get done. He asked them to sort out the higher priority items first to narrow down the list, and then prioritize within them.

The poor clients, who were already overwhelmed trying to keep the business running while trying to raise funding, were even more overwhelmed. It started out OK as a handful of cards were obviously not urgent. That left them with about 40 cards to sort out, most of which they felt HAD to be done in the next two months or sooner. We worked through the ones with question marks, attempting to clarify what was being asked for. Often it was simple. Sometimes it was "we'll just need to hold off on that one. I'll need some time to think about it and then get back to you." Occasionally our estimates were challenged, and on one or two occasions it got ugly.

The bottom line was that, in a few hours, we had a plan that everyone agreed to. Some tasks had been reassigned. We also had a few action items. One of them would keep a master list of the tasks, and if others were added, they would have to explicitly say where that task fit in the list of items and what would slip out.

We wish we could say that communication immediately improved and that XP got accepted and we all lived happily ever after. It didn't happen that way. However, for the next two months, everybody did a pretty good job of living in reality, whether they were happy about it or not. We did pretty well at hitting our estimates, and that was acknowledged as a good thing by the client.

By the way, about a year later, the founder called up Ken and asked forgiveness for how he had treated him. He recognized that he had often not been thinking clearly due to lack of sleep and was under a lot of pressure, but that it wasn't an excuse. He shared that the work we had done, especially the work Duff and Ken did together had stood the test of time and he had grown to appreciate it more and more. Did they become an XP shop? No. But they hired us to do some more work and didn't have an issue with us doing it in "XP fashion".

Welcome Back to Reality

At our first large XP client, we had a problem that no one from the business side seemed to be that concerned about an end date. They enjoyed steering, and understood that everything had a cost. They would ask for something new and we'd give them an estimate. Most of the time they would say, "OK, it's worth 5 days" or "N days". Our iterations were released internally. People liked progress but we had a growing concern that the scope creep was going to keep us from getting the product out externally.

All of the old stories were kept around, but nobody was really checking whether they were still necessary or whether the estimates on them were accurate or not.

The question was asked of management, "Is there a date we need to target for shipment release 1.0?". Although there had been no firm commitment to the market yet, they did feel that they should set a target. "March 2001". OK, then what we need to do is resort all of the stories and figure out what goes in the release.

The big "recommitment meeting" was planned. Before we all showed up, the designated customer and the project manager went through the list of stories and made sure they were all still relevant, and the customer came up with a few more that hadn't previously been captured. Then the big day(s). We planned several of them because we didn't know how long it was going to take until we had a new plan together.

We felt that the first thing we needed to do was to make sure everybody understood the gist of each story. So, we started at the top of the pile and worked our way down. For each story title, if anyone had a question about what it was, they would raise it. Many passed without explanation. Often, if someone asked for clarification, a one or two sentence answer would be sufficient. For about a fifth of them, someone would recognize an inconsistency with a previous story or an

overlap. Every once in a while a new one was added because of issues that were raised. After the better part of the day, we felt we had a pretty comprehensive list of remaining stories: over 200 of them. It was clear that we wouldn't get all of them done in the three or four iteration we had before we had to get it to system test (in this highly regulated environment, this hand-off was not negotiable). Time to prioritize.

We started by redefining the three categories of stories:

High - We don't have a marketable product without this.

Medium - A significant portion of the market would not buy the product without this.

Low - Nice to have. Some potential customers would like it, but it isn't likely to be the thing that will make or break their decision.

So, we started at the top again. The plan was for the customer to declare whether they were *high*, *medium*, or *low*. She could ask for clarification to help her make her choice or break them up into parts. E.g. We had a "user must be able to configure the system" story. It was clear that this was a *high*, but we could get by with allowing them to only configure a couple of pieces of the system necessary for it to physically work in their environment (a smaller story), and then make the remaining things they might want to configure a *low*. If the categorization the customer made didn't make sense to someone on the development team they could challenge it. It was possible that the development team didn't understand it, or the customer had a different picture of what it meant than the developers did, or that the customer wasn't thinking clearly. After being challenged and the ensuing discussion, the customer still had the final word about what category it was in.

This took the better part of another day.

At the end of it, the project manager said, "OK, now we estimate.". Ken interrupted. "May I make an observation? We could certainly pass these cards around and estimate each one of them, but I think there is something important we should not ignore before we do that. I'm holding about 50 high priority cards in my hand. We've all heard what they are, and there are very few one or two day stories in here. Let's say that the average story size in here is six days, which I think might be conservative but is probably in the right ballpark. Do you all agree?" (nods and other forms of affirmation). "OK, so if it's in the right ballpark, we're talking about 300 days worth of stories. Over the last few iterations, we've been pretty consistently hitting around 50 days per iteration. Therefore, the chances are slim to none that we can hit the target date even if we only stick to the *highs*." "So, are you saying we shouldn't estimate them?" "No, I'm saying that we shouldn't fool ourselves into thinking that there is any way that target date is realistic, especially since we've already been told we can't add any more resources to the project. And I've heard the

customer say that she didn't want to settle for just the highs in release one, but wanted to get at least a good chunk of the mediums before it got out there." (NOTE: The client has an existing product out in the field that this is supposed to replace. The customer felt it wouldn't be good to introduce a new product with less functionality even though it had a lot of other features that their old product did not have).

There was a lot more ensuing discussion, we'll save some of it until a later chapter. The point here is that there were some people living in a fantasy world (or at least hoping that the reality they were suspecting wouldn't unfold). Once they had the stack of cards in front of them, they were all living in the same reality.

It's a Bigger Project Than I Thought, But We Can Get Something Out Quicker Than I Thought

A long term client of ours had previously used us mostly for prototypes and sanity checks while they were exploring entering some completely new markets in the health care industry. They had spent a long time exploring a lot of different angles to entering the new market. It was tough to wait while we were all confident that the longer they waited to start something, the better the chance that someone else would beat them to it. Finally, one day the phone rang.

"Ken, I think we might finally be ready to go forward. Can we spend some time talking about what we want to do and come up with a plan to get there?"

"You bet. Thursday is the only day I could do it this week. If that doesn't work, we can try to juggle my schedule next week."

"I'd really like to get going on this. I can meet you Thursday afternoon. At least we can get started even if we don't finish it, we'll at least have a better feel for it."

The setting was perfect. The client, Ken at the studio with others to draw on if necessary, and a $1 shrink-wrapped CASE tool.

He cut open the tool and gave the client a 15-minute training course in "The Planning Game" and the use of the CASE tool. The client said, "I think I get it, but I'm not sure what kinds of things you want me to write on the cards or what granularity of detail you want me to write down." Ken said, "I'll tell you what. Why don't start by describing the things you want the system to do. I'll take notes by writing stuff on the cards and ask you for clarification as we go."

In the next couple of hours, we talked and recorded about 40 cards. Ken then asked him to prioritize. Then Ken said that we would start to put ballpark estimates on them. He asked for some clarification on some of them and quickly realized that some would take quite a while, so he asked how they might be broken down further into "minimum necessary functionality to demonstrate the capability", "functionality

necessary to really add value" and "nice to haves". He could put rough estimates on most of them, but there were a lot of questions about the possible user interface approach.

So Ken called a couple of the other guys in and we discussed some user interface issues. He seemed to want a web interface. We discussed a couple of approaches to that and raised some performance concerns due to the environment he expected the users to be working in. (On the go with a basic portable and intermittent connection… often via a phone line).

We ended up with about 60 cards with numbers on them, a few of which were big unknowns. We adjusted for the big unknowns by adding a couple of 8 craft unit (similar to an "ideal engineering day" from XPE) cards that said "revisit [X]" since we suspected that the first thing we tried would not be sufficient but would uncover issues. Ken asked him how fast he would want to move in terms of how much could they spend per month. He said, "I'm not sure, but let's go with $X/month". "OK, how long do you want each iteration to be: 2, 3, or 4 weeks?". "Let's go with 4. You know our company, there's no way we could respond to something you produce any faster than that."

"OK. That means you get 12 craft units per iteration. The numbers are on the cards. Start with iteration one and pick which cards you'd like to tackle first. Once you've got 12 craft units, start iteration two, and so on."

In about 15 minutes, he had 13 piles (the 13th being only partially full). Based on some of the things he had deferred in our earlier discussion, Ken told him that he might want to be conservative and assume the whole thing would take 18 months at that pace. He could shorten it by picking up the pace, but he probably couldn't make it under twelve months without losing a lot of efficiency if he could get it shorter than that at all.

"Wow. This is a lot bigger project than I thought. And you are pretty confident in your numbers?"

"As sure as I can be without getting into a lot more detail. Based on my experience, I'd say we're not off any more than 50% to get something out that does all that you've asked for in some way. That's why I suggested figuring 18 months even though we had just over 12 iterations here. But let's face it, we're just guessing at what this is really going to look like. We should probably work on the first two or three iterations and we'll have a much better clue.

"Would what we put in those first three piles be enough for you to show it to someone and get some real feedback?"

"Yeah. I'm pretty sure it would. I'm pretty excited about that. I was hoping we could get something to show people in three months, but I was doubtful that it would

be possible. That's really good. It might make the whole project easier to swallow if they know that they can have something to shop around in 3 months."

"OK, where do we go from here? (tongue firmly planted in cheek) Can we start Monday?"

"I wish. But I think I have enough to go on to present this to some people next week. Let me write down the stories and the time frames. I'll turn it into something more polished and present it to some senior management people.

"I was real impressed with this process, but I don't think they'll give me that much money if I come into there office with nothing but a handle of cards."

"Go figure."

"But really. I can't imagine any other process I've seen giving me this much information in this short amount of time. It's pretty powerful."

"Yeah. It's powerful because it's simple and there are no smoke or mirrors involved."

By the way, they still haven't gotten the funding for that project, but they were back the next month to scope out a different project in half a day. That one worked, too.

You may not be able to do as good a job of Ken in getting this all down in half a day. On the other hand, you may be able to do a better job. If you have a process that sets reality in front of your eyes in an easy to manipulate way please let us know what it is. If you have one that works as good and is cheaper, let us know that, too. Otherwise, use the $1 CASE tool.

The Xtreme Hour

Several years ago, somebody [look up who] came up with the concept of the ExtremeHour. The idea was to get people used to the idea of many of the concepts of XP.

Recently, we've been making pretty good use of this. We've modified it from its original form slightly, but it fundamentally is the same.

The premise is that you divide into groups of 6-10. Part of that group plays the role of the development team, and part of it plays the role of the business team. Together they are tasked with building a product from concept to delivery in one hour. (Really they just deliver a picture of the product in one hour).

You can do this with people whose natural roles are business or technical or a combination. It is ideal when you have a mix of these people and you make the business people play the role of development and the technical people play the role

of business. That way they get to walk an hour in the other's shoes. But, the point usually comes home whether or not you mix them up this way.

At the end, ask them what they've learned. Here is the paraphrase of a VP of IT Operations at a large multinational bank:

"I learned how important it is to have more collaboration and to spend more time communicating. I also learned that when I ask for something, that I should specify whenever possible parameters for testing how they will know that it works. In an environment like this, I can clearly see the confusion that vague requirements make and how hard it is at times to figure out how to do something even when the requirements are clear because it has to work with all of the other requirements and often, they seem to be contradictory or at least difficult to implement such that both requirements are met."

Would you like to hear someone in your management chain say something like that. Ask them if they've got an hour or two to learn about the development process.

Steering

Without a direction, you can't begin to implement XP. If you don't plan, you will have no idea what to code today, or tomorrow, or next week[2]. That means the "programming" piece of XP won't matter. A great system metaphor (of any of the other practices we talk about in Section Three) without a roadmap for making it real is worth about as much as sock lint.

As Kent Beck pointed out in XP Explained, XP is like driving a car. Planning is like steering. We make small adjustments to keep ourselves on the road in our lane. When we end up in a ditch, planning is how we get out of it. If you don't get good at steering, you might as well not drive.

When Roy was in college, he loved to watch the men's crew team eat. These guys burned calories standing still, so they had to gorge themselves often just to get enough calories to survive. Since nobody else in school was quite this weird, the crew team ate as a pack. They would sit at a big table, heads down, attacking their food like hyenas at a kill. Every once in a while, one guy would lift his head to look around and check the environment. We work and plan like that.

Planning and short iterations (Chapter 11) work together here. We start by planning. We lower our heads and go for broke for a short time. Then we lift our heads up and plan some more. We learn from what we saw during the last sprint. We fold the lessons into the revised plan. Then we're heads-down again, sprinting through the next iteration. We never go too long between planning sessions, usually

[2] Pick up a copy of *Planning XP* by Kent Beck and Martin Fowler for more eloquence than this.

two to four weeks. The plan isn't reality, it's just a guide. The act of planning is the important thing. So we do it so often we that we feel uncomfortable when we go too long without it

Embracing Change Fearlessly

As we said in Chapter 6, XP is dead without trust. It requires brutal honesty all the time. The planning process in XP lets customers and developers learn their roles, and gives them a forum for being brutally honest with each other. The customer and the developers know their roles. They act them out every time they plan. And they plan all the time. This takes fear out of the equation. It makes a change an opportunity to learn, to do the right thing, and to make the results of the project better faster.

Most people have lived outside the XP world at some point in their professional careers (perhaps you still are, poor soul). Think about the interactions between customers and developers without XP.

Does this exchange sound familiar?

CUSTOMER:	"We need to segment our customers differently, based on the new marketing strategy. The old way just won't work anymore."
DEVELOPER:	"That wasn't a requirement."
CUSTOMER:	"It wasn't a requirement when we started, but it is now."
DEVELOPER:	"Well, put in a change request, but I can't make any promises."
CUSTOMER:	"@#$%#* !!!!"

Where did this change request idea come from? It came from developers who got tired of being asked to make changes without being given the time to make them. Why weren't they given the time to make them? Business people would learn new things were important, but they were never given a system that would allow them to easily trade one requirement for another. The development process was just a black box with one input and one output with no controls on it. This system pits one side against another. It's foolish. You can't play to win with this kind of system.

You've got both sides playing not to lose. Development can't get started until the business people are satisfied they have enough input to do it right. Business people aren't confident they have it right, so they don't let developers start. But the market demands a product now. Developers finally get started and they are already under the gun. Under pressure like this to produce everything quickly, why would you expect anything other than them making it hard for business people to make requests that impact their time.

Change is a given. Most methodologies are set up in a way that makes customers and developers fear change. Fear kills projects. Customers fear not getting what they want, and not being able to change their minds as requirements change. Developers fear not having a life, and being whipsawed all the time as requirements fly all over the map. This is a recipe for conflict and distraction.

You should know by now that the horizon changes as you travel, so trying to design everything up front is nuts. Well, requirements are the customer perspective on design. It is the only "design" customers really know, or care to know. Expecting customers to be able to know all the requirements up front is just as silly as expecting developers to know the whole system design up front, but this happens every day.

Developers don't fare much better. They are like the British Light Brigade in the Crimean War. They charge into battle knowing full well that they'll be butchered. They are coding obsolescence. They are producing code of questionable quality that nobody really likes. It might not even be used. And they feel like they've been shot to pieces by all the overtime heroics they put in trying to do the impossible.

XP is different, because its approach to planning is different. *XP Installed* used the analogy of the "circle of life" to describe how work gets done in XP (we've amended it slightly):

- The customer defines business value and creates stories to translate it into system function.
- Developers estimate costs for each story, usually in some unit of time.
- Developers tell the customer how fast the team can move (the team's velocity), which dictates how much work can get done per iteration.
- The customer picks a group of stories to do next, with estimates that add up to the development team's speed for an iteration.
- Developers build those stories.
- The process repeats.

Developers get to produce great code. They get to estimate effort and stick to their estimates (customers can't quibble). They get to tell customers how much they can eat at the buffet, given the available time. They don't get slaughtered.

Customers get to set priorities. They get enough information to make intelligent decisions about what to do now and what to defer. They get to change their minds. They don't feel locked in.

XP lets you move past the petty wars over turf and CYA management (which are really other words for "fear") to getting things done.

How To Plan

You're always wrong about the future, even if you get the major events right. Unfortunately, you're measured on how close to right about the future you were. The further off you are, the more trouble you're in with those who are doing the measuring. There is no way to fix this situation.

The only thing you can do is go with what you know for sure:

- Time keeps moving, so you can predict that a particular date will arrive in a certain number of calendar days.
- Before that date arrives, if everyone involved agrees to dedicate a certain number of days to a particular set of tasks designed to solve a problem, you can make progress toward the goal.
- If you don't apply effort toward achieving the goal, we guarantee you won't be any closer to it by the time the date arrives.

So plan. Reality won't ever match your plan, so don't spend too much time on it (a day or two per iteration feels about right). Specify the target, estimate the effort in "ideal days", then execute the effort. When you're done, take stock to see how far off you were when you estimated. The next time around, you'll have more experience to guide you. This is common sense, but we've noticed it isn't that common. "Fire! Ready! Aim!" isn't as backwards as it sounds.

The Release Planning Game

One of the refreshing things about XP is that it isn't unnecessarily formal. Planning is a "back to basics" affair. The customer and the developers sit a in a room with whiteboards and note cards. The customer starts talking about what the system needs to do. The customer writes each requirement (yes, these are requirements) on a card. Try to keep the stories small enough so that one developer (with a pair of course) can accomplish them within one of your iterations. Developers ask clarifying questions. The customer answers them. If a developer thinks there is a story missing, he can suggest it, but only customers write (or at least validate) stories (and they can reject suggested stories entirely if they want to). That's step one.

Once the cards are done, the grand sorting begins. The customer sorts them into three piles based on business value:

- Necessary for the system to be viable.
- Not absolutely necessary, but valuable
- Nice to have

Then the developers make their estimates. NOTE: it doesn't need to happen in this order, but people usually care more about the estimates for the first and second priorities. Identify how sure you are of the estimate. If you need some time to do some research, identify how much time you need to give them an estimate you can have any confidence in. Don't do this for every story or ask for an inordinate amount of time. Half a day is usually more than enough… a full day is usually more than sufficient. Remember, you are not looking for a guarantee, just an estimate. If they won't give you the time, either refuse to give them an estimate or make it big. Don't give them anything else.

At this point, and again in iteration planning at a much smaller scale, the development team will explore the stories they have no clue how to estimate. This is called…drum roll, please…Exploration. And there was much rejoicing. We use Exploration to accomplish a few critical things:

- To lower the risk of making a bum estimate
- To experiment with various implementation options to increase and demonstrate our understanding of the problem
- To determine our velocity for implementing stories

Exploration can last anywhere from one day to two months, depending on the size of the project, how well-defined the stories are, and the issues surrounding the technology choices. If we are just starting a brand new project that is not extremely similar to one we've done with the same team members before, the typical length is about a month if all the necessary resources (human and otherwise) are readily available.

When Exploration is done, and it shouldn't last very long, the developers move on to estimating the stories. Developers estimate coding time for each story in what we call *craft units*. Each one corresponds roughly to a half-week's work for a single developer or an "ideal day" (days where all the developer has to do is work on the task, without distraction). A story that will take a week gets two units, and so on. (Your mileage may vary). If a story needs less than one craft unit, it gets combined with another story for estimation purposes. If a story spans more than N deliverable units (where N is the number of units one developer can do in an iteration), the customer split into smaller craft units. No story may span more than one iteration worth of deliverable units (eight for us on a four week iteration. We prefer smaller iterations… see next chapter). If it does, the customer splits it into smaller stories.

Given the estimates from the developers and the velocity they settled on during initial planning, the customer has a choice. He/she can either

- choose a set of cards, and base the completion date on the estimates on the cards, or
- set a completion date, and include the highest priority cards which fit based on the estimates

The stack of cards you've got is your next Release. Congratulations.

This process is done at the beginning of each release. It should take no more than a few days, excluding time for necessary spikes, and usually takes less. When it's done, the customer has a set of cards for the next release, a price tag for each one (the estimate), and an idea of what will come in future releases.

The Iteration Planning Game

We left off with a stack of cards that need to be done for a release. This is too much to concentrate on all at once, so the customer sorts that bigish stack of cards into iteration-sized piles, based on two things:

- business value and anything else that determines priority for the customer
- the velocity of the development team

Now you have to plan the next iteration. Don't worry. Iteration planning is just release planning in miniature.

Reserve some time at the beginning of each iteration (say, one or two days) to review the last iteration's work. Be critical. Take steps to reduce business and technical risk based on what you learn and changes in the customer's needs. Then use a downsized version of the Planning Game to break the stories down.

This time, however, the development team breaks the stories down into "tasks". The act of breaking them down into tasks is actually a design exercise. You are figuring out the design approach you are going to take to implement the stories, and each step is a task. How small should you break up the tasks? Two days is the maximum we feel comfortable with. I've heard others say one day or one-half day. Go smaller until you are better at estimating large chunks. Add up the chunks and revise the story estimates for the customer.

It usually takes us a day or two to nail this down. So iteration planning usually goes something like this:

< 1 hour - Demonstration and discussion of what we got accomplished. There should be no real surprises here if your customer has been active in the iteration, it is just a syncing up of what we did and did not do. If they haven't been active, you might need more time for this.

1-2 hours - Discussion of what the next iteration should contain based on what he had thought we'd tackle next and what we should tackle next based on what we know now… this results in a candidates list which is prioritized by the customer.

4-6 hours - People volunteer to lead the breaking down of each of the candidates into tasks. This can take lots of forms, but basically we break into small groups and do it, sometime with the customer and sometimes without. Sometimes, an individual does some quick exploration, and then calls someone over to work through it. Sometimes the customer is grabbed.

Next day - we have a list of tasks and possibly a list of issues that need to be resolved. People sign up for the highest priority tasks and start them. We review what we think we can get done with the customer and sometime during the day we're done iteration planning (or at least 90% of it). It's kind of fluid as to when the planning ends and the doing begins.

If, after breaking down the stories into tasks, you find that your estimate for the iteration is bigger than you had originally estimated, share it with the customer. Tell them how many days of tasks you are over. They might pull out a story that is that many tasks (after they are done being disappointed). On the other hand, they may want to re-split a story. Now that they are broken down into tasks, there might be natural, obvious breaks in the story that weren't obvious before. The remaining tasks can go into a future story… probably next iteration, but that's up to the customer when we start the next iteration.

In "Planning", Kent & Martin suggests only committing to the number of stories you delivered last iteration. That's not a bad rule. We certainly take last iteration into consideration when we plan the next iteration. However, we also adjust per developer. A new developer may actually be able to contribute based on their skill level, their familiarity with the domain, and how much they've worked with other members of the team before. We also adjust for whatever else is going on (vacations, other known assignments, whatever) that we are confident will have an impact. We tell the developer how many units we think we'll be able to deliver based on all this.

We weren't very good early on. Sometimes we were off by 35%. However, we've gotten pretty good lately, and we seem to be within 5-10% of actual units delivered. This should not be confused with accuracy of estimates on individual tasks. We're getting better at that, too. See the Tracking section for more on this.

When Exploration is finished, get everybody together in a room again and take to the whiteboard. The development team brainstorms the tasks for each iteration. Our rule of thumb is that any story worth two or more deliverable units should be broken down into smaller tasks. The customer sits in to answer questions to listen to make sure the developers understand the stories. If the customer hears somebody say

something that indicates the team missed the point, he/she pipes up and steers a little. Once the tasks are on the board, the developers hold a kind of auction (it can get a little funny). Somebody chimes in to take ownership of the task and estimate it. Other developers (not customers) can take pot shots at the estimate to refine it.

When the ownership/estimation auction is done, the developers review the estimates for each story and revise as necessary. If something needs to fall off the plate for the current iteration, the customer picks what falls off. If there's more room than there used to be, the customer picks what gets added. Update the plan if the changes warrant it.

At the end of each iteration, the development team commits to the system being able to run with the functionality described in the stories for that iteration. The initial iterations probably won't produce a system that does much of anything useful (although sometimes this isn't true). They are still important as a gauge for customers to be sure they are getting what they expect out of the development process.

This iteration planning happens at the beginning of each iteration. It usually takes anywhere from a half-day to two days, excluding time for necessary spikes. It shouldn't take more than two days. When it's done, developers know what they'll be working on next and how long it should take. They have their marching orders.

What if you don't have a single customer? Travis Griggs speaks of unique approach they've come up with called "The Senate Game" [need to insert Travis description here]

What Plans Are

Planning is the way we steer an XP project. We use stories to sketch out where we would like to go. We defer specifying details for stories until the iteration in which we'll code them. We make small adjustments as we go.

Planning minimizes your chances of ending up in a ditch, and can help you get out one when you're in it. Plan like people were said to have voted in Chicago in 1968: early and often.

If you don't have a map you'll get lost. If you never deviate from the map, you'll probably miss lots of good stuff that is off the beaten path. Thus, we come up with a plan but we're willing to stray from it.

Paul Grobstein said that the concept of being "right" should be replaced with that of being "progressively less wrong." The former measures your success or failure by proximity to a fixed point, a target, set at the beginning. This encourages people to put on blinders. The latter measures your success or failure by charting your progress from your starting point. This encourages exploration, respect for

experience, and the appreciation of the value of each individual's perspective on a problem. XP planning is all about learning, and then applying the lessons learned as soon as possible. We hope to be "progressively less wrong" over time.

How To Start Planning

You'll never know all there is to know about planning and estimating, but you won't know anything if you don't get out there and do it. Remember that XP is about learning and doing it better next time. The more you plan, the better you'll get at it.

When you're just getting started, it's more important to get used to that rhythm than it is to be right. At the beginning, you won't have a clue what your team's velocity will be. Guess and refine it later. You'll do this frequently when you start. Get used to it. Don't be lazy, but don't be too tough on yourself either.

One of the people in Ken's 1999 OOPSLA Workshop "Refining the Practices of Extreme Programming" put it this way:

> *Starting with nothing more than a very high level statement of functionality to be implemented that month, the first task was for the team to break this down into more manageable pieces. Our intention was to write on a whiteboard a list of sub-requirements, each of which could be implemented by a team in on half-day. This effort was largely a failure. We did identify functional pieces that could be doled out to the pair teams, but our accuracy in identifying half-day tasks was way off….We never did get to the goal of actually identifying half-day tasks reliably, but soon they were being completed in a whole day with very good consistency. [Rob Billington, submission to "Refining the Practices of Extreme Programming", OOPSLA 1999]*

That's the spirit. When you were learning to drive a car, steering was a tough skill to pick up. You probably swerved a lot. After a while, you didn't have to think about it as much – it started to become second nature. Planning is the same way.

Start by having your customer write cards. Have them put each feature of the proposed system on a card. This might be uncomfortable for them at first, but it's the same exercise they would be doing to help you create a requirements spec. You're just giving them the power to write it down, and you're making each feature of the system a discrete card.

When it's time for story estimation, use a few rules of thumb until you have enough empirical evidence to refine the numbers for your team:

- Each story should require no fewer than one half-week for a single developer. If it does, combine it with another.

- Have the customer split any story worth more than a week for a single developer.
- Start with two-week iterations to increase the amount of feedback you get early in the process. You can lengthen this later, if it makes sense.
- Guess at a velocity of four days per iteration to start. Adjust upward in the next iteration if you end up getting more work done than that.

Once the customer has specified what is to be included in the release, based on the team's velocity, have him sort the cards into iteration-sized piles. Then have developers brainstorm tasks for each story, estimate them, and accept responsibility for them. Some additional rules of thumb here:

- Estimate tasks at 0.25 to two days in the beginning.
- Never fight a ground war in Russia in the wintertime.
- Never match wits with a Sicilian when death is on the line.

Remember, the goal with planning is to keep it simple. The best way to do this is keep it ridiculously short in the beginning. Give yourself lots of feedback, and lots of opportunities to steer. As your skills improve, you can begin looking a little farther down the road as you drive (but never too far).

The Art of Estimation

The bedrock of planning is estimating how long work will take. Stories and tasks are great, but eventually the rubber has to meet the road. Estimation is a strange thing. It's guessing based on our experience. The more experience you have in a given context, the better your estimates will be. If we've done exactly the same thing before on the same project, we use what Beck and Fowler call "Yesterday's Weather" to make our estimates (that is, we just assume the same estimate this time around). If we haven't, we look for similar experiences on the same project and estimate from them. Only as a last resort do we just guess.

That said, estimation can be tough. This is especially true when people are working with new technologies and/or in new environments. Tell people to keep it simple. Break stories down into very small tasks (remember the guideline of 0.25 to two days we mentioned in *The Iteration Planning Game*). If the sum of the days for the tasks in a story add up to longer than a week or so, have your customer split the story. This exercise will help people do a better job at two things:

1. breaking down stories to make sure they know what will and won't be accomplished

2. making accurate estimates

Like we said, this is equal parts art and skill. It will take a while to get good at it, but you will improve with practice. On a recent project, Ralph Johnson made the astute observation, "There is nothing that you can't do better the more you do it." Your estimates won't ever be perfect, but you can hone your skills. Before you know it, people will start to take your estimates and plans seriously as they get closer and closer to reality.

What's Velocity?

Estimating is the art of determining how big some effort is. That's not worth much unless you also can tell your customer when they'll get it. That's velocity.

Velocity tells you how fast your team can produce the stories required by your customers in a single iteration.[3] The Planning Game and the Iteration Planning Game tell you how much everything costs. The Tracker or the Coach or the Manager then specifies the velocity of the team, based on past performance. No other basis makes sense. Beck and Fowler describe this as "Yesterday's Weather": assume new stuff that's like stuff you've already done will take just as long to do. If your team delivered seven stories in the last iteration, assume they'll do the same the next time around. It's that simple.

There are several ways to express velocity, but they all answer the same question: how much can your team get done in a single iteration? The two major candidates appear to be these:

- ideal days
- some sort of "story points"

The reason these sound different is that they are based on different inputs. The ideal days form says that your team can produce a certain number of ideal days' worth of stuff in an iteration's worth of calendar days. The "story points" approach says that your team can produce a certain number of points (a simple way to express "bigness", or difficulty) in a single iteration. Six of one, half-dozen of the other, really. Pick one and go with it.

---

3 "Speed" is probably a better word to use. Sorry, Kent. Velocity is a vector, while speed is a scalar. What's being described is how fast you are coding stories. Direction isn't addressed, except to say that you're moving forward by adding system features. But, hey, we're not being picky.

We express velocity in terms of deliverable units per iteration (DUPI), which is our version of story points. As we said earlier in *The Planning Game*, each deliverable unit is about one-half a week of ideal days for one developer. No story may span more than what a single developer could do in an iteration. We assume a load factor of 2.5. That's our starting point for all projects. We can adjust up or down based on the individual project, once we have some experience in that environment.Given those inputs, each developer can handle roughly 8 DUPI in a four-week iteration, or 6 in a three-week iteration

This approach standardizes our measurements. The size of each story is a multiple of a known quantity. There's lots of ways to express it:a single deliverable unit, an ideal day, or "about half a person week". Load factor can vary by person, but we usually state velocity in terms of the team as a whole. We tell the customer our collective velocity: for example, 32DUPIs per iteration. When we start an iteration we figure out how many days of the iteration each developer is expected to be present, take into account other distractions they'll have to adjust their individual load factor, and total up the days we expect that iteration. Here is an example:

Developer	Days Available	Load Factor	Deliverable Units
Joe	15	2.5	6
Nathaniel	18	2.5	7
Ken	10	2.5	4
Amy	19	4	5
Duff	16	2.5	6
Donna	20	3	7
Totals			35

When we estimate, one of us throws out a number of ideal days for a story, we discuss it a bit and put a number up. If the stories add up to more than the available days, we cut some out

Beck talked about "load factor" in *XP Explained*. Load factor is the multiple you use to get from ideal programming days to calendar days. If you can get eight ideal days of work done in a 20-day iteration (4 weeks), your load factor is 2.5 (8 / 20). In *Planning X*P, Beck and Martin forsook load factor somewhat, and claimed velocity is the only measure you need. We tend to agree, but we think it's nice to be able to derive both, just in case you have to estimate how long a short-term project will take.

Estimation Rules?

Bill Wake says 1/3 story per week per developer and tasks should be less than 3 days . Ken says estimate a load factor… it's not that complicated. Kent says "use yesterday's weather"? What's best?

You just gotta find something that works for you and recognize that "estimation" will NEVER give "exact predictions".

Chapter 12

Thinking Short

An idealist believes the short run doesn't count. A cynic believes the long run doesn't matter. A realist believes that what is done or left undone in the short run determines the long run.

-- Sidney J. Harris

Keep your iterations short. This minimizes negative surprises and keeps you on track. Release early and often. This will tell you if the track you think you should be on is what your users really want. Both short iterations and small releases reduce your project risk.

Most other software development force people to predict the weather for the next six months to one year. We don't know about you, but we won't leave our umbrellas at home if these folks tell us to.

XP takes the opposite approach. We think short. This means two things:

1. develop in short iterations
2. release often

We do everything a little bit every day (remember the rhythm of code a little, test a little, integrate a little), and we take stock of what we've done at frequent intervals. That's an iteration. We also release a working system to end users frequently in order to get real-world feedback on how we're doing.

Short iterations keep a project under control. They provide natural, regular, and frequent checkpoints for planning. This lets you make small corrections instead of over steering to try to get out of a skid.

Short iterations also keep the people on the project focused on what counts – delivering business value to customers. If you don't do this, nothing else matters.

Short releases keep you grounded in reality. There is nothing like feedback from real users to keep you developing the right system.

Maintaining Control

The United States Air Force Blue Angels squadron of elite pilots tours the world giving death defying demonstrations of aviation skill. They rarely make a mistake. When they do, it's catastrophic. In one incident, the entire squadron followed the lead pilot as he flew into the ground.

The Blue Angels have a special investigative unit that tries to get to bottom of incidents like this. They look at what they call the "error chain" to figure it out. The lead pilot lost his focus during the maneuver. He lost his focus because he was tired. He was tired because his back hurt the night before and he didn't get enough sleep. His back hurt because he wrenched it playing volleyball a week ago and didn't get the proper medical attention. The whole squadron died because of one small error that one person made long before the final catastrophe.

Software projects are like that. The longer you let them go without making small corrections, the more those small errors snowball. Pretty soon, you're out of control and the whole squadron flies into the ground.

We said in Chapter 10 that you'll always be wrong about the future. Why is it that projects continually ignore this reality? The longer you go between measurements of how you're doing, the more likely you are to crash. Keeping your iterations short makes this rare (although not impossible). You get feedback on how you're doing in weeks instead of months. You get it sooner rather than later, while there's still time to do something about it.

Staying Focused

The longer the time between measurements of where you are, the more pressure you will feel to make it look like you were right when you predicted the future. If you predicted a lot over a long period of time, it won't take too long until you are overwhelmed by all the crises you have to handle and you won't know where to start fixing them. You'll lose focus because you are worried about spinning the story about why you weren't really that far off in your predictions.

What if you took stock of where you are every few weeks? If you did that, you wouldn't be able to afford to lose your focus on delivering business value to your customers. With only two or three weeks before your next measurement, you don't have time for distractions. Best of all, you'll be more likely to predict accurately, because you don't have to predict the weather months in advance. That means you won't have to waste your time on spin.

You will encounter surprises. Some will be negative. Measuring often minimizes the number of those surprises and decreases their magnitude. When you encounter one, you can tackle it and move on. You don't have the false belief that

you can put it off, work on something else, and get back to it before the next milestone, because the next milestone is only a few days away.

Staying Grounded In Reality

How do you know if you're developing the right system? Ask the users. The problem with many development methods is that they

- assume what users want, or
- they ask users what they want at the beginning and assume they knew what the users meant

Both are dangerous. Either way, you will be wrong. Communication is a tricky thing, as we discussed in Chapter 10. This is especially true when you are talking to users who might not know exactly know what they really want. Users rarely know what they want because they've never seen it. If they had, they probably wouldn't need you to build software for them. Giving them something to try can help them imagine what they really need. It's better to get something in their hands to let them explore, then update and refine constantly.

In his paper Improving Software Quality, Bob Green says,

> *Until you deliver a program to the client, you have not accomplished anything, and you haven't started receiving the objective feedback that will ensure a quality system. The advantage of going after the immediate problem first is two-fold: it gets the client on your side, and it uncovers facts about the problem that may make the rest of the project simpler or unnecessary. – Bob Green, "Improving Software Quality" (http://www.robelle.com/library/papers/quality/)*

Deliver something. It doesn't have to be complete. It doesn't have to do much at all. It just has to work. The more often you do this, the more often you get the feedback you need in order to steer the project toward what users want. You also get to show your customer that you are making progress toward that goal.

For most systems, you have almost no chance of guessing right about what users really want. The secret is to guess wrong early and often. Small, frequent releases let you fail fast and learn from the last failure. The result is a product that is as close to what users want as you can get.

How To Think Short

Decrease the space between your predictions and reality. The best strategy for doing this is to keep your iterations short.

Early last year, Ward Cunningham visited us to see how we were using XP at one of our clients. While he was here, Ward paired with several members of the team. One person recalled how amazed he was at the very small steps Ward took when he was coding. But they worked. He had almost constant feedback. Iterations are that principle writ large.

In general, the shorter your iterations, the more focused you will be. Make your iterations long enough to do something that matters to your customer, but short enough to avoid a crash. This could be a week in a highly productive environment with highly productive developers on a small project. The more you stray from that model, the longer your iterations should be. Two or three weeks is, closer to two, is what Beck and Fowler suggest. We use four right now on at our largest client, but we're looking to shorten that.

Once you get into habit of running these small iterations, get into the habit of releasing early and often. This is uncomfortable for lots of customers. They don't want to release something until it's "done". Emphasize the fact that doing this ensures that the system will be done wrong. Focus on producing the most valuable stories first, so that each release, not matter how small, does something that the customer values most at that point in the project. Then see if users agree.

What happens if users don't like what you released? That is a good thing if you are releasing early and often, because you still have time to do something about it. If you wait until the end of the project to have your grand unveiling, you have only one chance to give the users exactly what they want.

How To Start Thinking Short

There is no way to start working in an iterative way gradually. Try it out. Grab two weeks worth of stuff from the stack of stories you came up with in your first planning session and attack them. Set a hard deadline two weeks away and refuse to compromise on it. Then work for two weeks.

When your deadline comes, stop. Take stock of what you got done and what you didn't. Ask yourself some questions:

- How were our estimates? Were they consistently high or low?
- Did we assume we could move faster than we really could?

Once you've taken stock, go through the exercise for the second iteration. Make sure you fold in the lessons you learned. Pretty soon, this will become a habit. You'll start thinking in iteration-sized chunks of time.

The same goes for releases. According to Kent Beck, if you are doing the other essentials, this should get much easier[1]:

- If you're planning, you are the working on the most valuable stories at any point, so even a small system has high value.
- If you're integrating continuously (more on this in Chapter 16), packaging a release at any point is easy.
- If you're testing like mad (more on this in Chapter 13), you don't have to go through a lengthy test phase before you ship.
- If your design is as simple as it can be all the time, it will be sufficient for the next incremental release.

The biggest barriers to releasing early and often are developers who don't want to release something before it's "done" and customers who don't want users not to be impressed. Waiting a long time to release doesn't fix the problem, because you probably will produce something users aren't happy with anyway.

Think of small releases like an evolving prototype. The more revs you have, the closer your prototype can get to the reality you're targeting. It is the best way to reduce the risk of guessing wrong about user requirements.

What If Business & Development Have Different Ideas of "Short"

We've found that developers are more prone to lose focus if the iterations are longer than 3 weeks. We've also found that, for many customers, having a planning session more than every 4 weeks can be counterproductive, and micromanagement tendencies kick in.

At one client, we addressed this by dividing our four-week iterations into "half-iterations". We decide which half of the tasks need to be done first and focus on getting them done in the first two weeks. We have a natural checkpoint to make adjustments on what our tasks are and our confidence of our estimates. The "time pressure" is just enough so that we don't get tempted to go down rabbit trails thinking we have enough time to get back on track later in the iteration.

[1] Beck, K. *Extreme Programming Explained*, Addison-Wesley, 2000, p. 64.

Chapter 13

Write the Tests, Run the Tests

asdf.
--

> *Write tests before you write the code to implement them. This will feel odd, but you'll get better with time. Soon, it will feel odd not to write tests first.*

If you are like most of the readers of *XP Explained*, you are already convinced that having tests is a good thing. You also aren't in the habit of writing tests before you write code. You're probably a little intimidated or confused by the idea. We typically hear things like this:

> *How can I write tests before I've written the code I'm going to test? I don't even know what the classes are going to look like yet. They're going to evolve.*

Ken used to say stuff like this back in the 80s when he heard some proponents of Objective-C claim that each class should have associated tests that should be written first. He has now confessed and repented. You need to write tests first precisely because you don't know what the classes are going to look like yet, and because they are going to evolve.

Writing tests first keeps entropy from destroying your code. We write tests. Then we write just enough code to get the tests to pass, no more. That is the best way to keep your code clean.

Nothing gives you more confidence when changing code than having immediate feedback about whether or not your changes broke anything. That's what having tests does for you. Without the tests, you can't have confidence. Without confidence, code doesn't get changed, even when it needs it.

Better still, we have found that well-written unit tests are some of the best documentation possible. They communicate the intent of the code being tested better than any description on paper ever could. We can kill two birds with one stone.[1]

Keeping Code Clean

Writing the bare minimum code necessary to make the tests run keeps you from wasting time on extra features. These hooks you might need later usually have to change once to you get to "later," if they're ever used at all. But without the tests there to guide you while you write code, the temptation to design for the future is just too great.

Writing the minimum necessary code the first time around ensures that the refactoring you do later isn't a complete overhaul. You can't get it perfect the first time all the time. You will refactor your code. But start with the simplest code that could possibly work. If your *refactoring* turns into *redesign* on a regular basis, your velocity will tank. Moving at top speed is your bread and butter. Guard your velocity with your life.

Think of this like keeping a room clean. Roy's mother used to say that doing that is simple. Don't let the room get dirty. If you let the dirt build up, cleaning it becomes a much bigger job that you end up putting off. Pretty soon you've got an intractable mess. Roy's room is still always impeccably neat, by the way.

Confidence

In 1955, Dr. Jonas Salk administered his experimental polio vaccine to himself, his wife, and their three sons. In a newspaper interview, he said that he did it because he knew it would work. That's courage in action. Where did it come from? Salk himself said, "It is courage based on confidence, not daring, and it is confidence based on experience."

At least once or twice during most iterations on our projects, someone sees that they need to make a radical change to some aspect of the system. When they do, a bunch of tests break. This is expected. The pair works through the failures one by one, modifying code to get each test to pass. This could take a few minutes or a few hours. Invariably, when the pair is integrating the changes one member says, "Wow, can you imagine how hard this would have been to do without these tests?!"

XP depends on courage, not bravado or reckless daring. Having tests to run to confirm each change to our code keeps us from being reckless. It lets us proceed bravely, knowing that we can prove to ourselves that our changes work. Instead of

[1] No birds were harmed during the writing of this book. Keep reading.

holding lots of meetings to determine if a change will break something else, we can just try it, run the tests, and find out. It's hard to be bold in the dark. The tests turn on the lights.

Tests As Documentation

We have found that well-written tests are less painful to produce than other forms of documentation. Not only that, they're better.

When you write tests first, you don't have to loop back and write documentation later. It is an inseparable part of writing the code itself. It's sort of like having a certain dollar amount each month taken out of your paycheck and put into a 401K. After a while, you just don't notice it anymore. Any speed we lose in writing tests first, we more than make up in reduced debugging time and reduced documentation time. This is a great boost to your team's velocity in and of itself, not to mention the increased velocity you get from having more confidence.

Even if this weren't true, tests still would be worth the effort. They are better than any other kind of documentation we've ever seen.

Tests are more detailed than verbal descriptions on paper, without being cluttered by extra words that describe what the code syntax says. Think about it. The written code documentation we've seen wastes a lot of space saying things like "The getMeal(PotPie) method on Shopper asks a Store for a PotPie. It casts this as a Meal and calls addToBasket(Meal), passing it the Meal…" Why not just look at the code? When you have a test, you can see a discrete Assertion (in JUnit) that tests this very behavior and displays a message.

Tests also are more useful than other forms of documentation we've seen. They give you a scenario in which the behavior of the code should work as intended. Want to understand what a class does? Look at its tests to see exactly what you should expect in certain scenarios. Other documentation struggles to do that. It usually fails, if it tries at all.

Best of all, tests are absolutely up to date all the time (unless you're cheating). What other documentation can claim that? We remember life before XP. Nine times out of ten, whenever we walked onto a new project somebody gave us a document that supposedly described the system we would be working on. It was usually a very thick binder. As soon as they put this in our hands, they said something like, "This should help a little, but it's out of date. Come talk to me after you read it." So what earthly good was that document? Maybe it built character to put it together, but that's all it was good for.

Not all non-code documentation is bad. When a customer needs a written document other than code (say, for regulatory approval), you should produce it. XP simply says you should produce only the written documents that you absolutely

need. The problem is that traditional approaches tend to substitute documentation for communication, and tend to exaggerate progress with documentation. Producing a document may get you closer to your goal, but documents don't run. In the end, it's the code the counts. Alistair Cockburn says:

> *Written, reviewed requirements and design documents are "promises" for what will be built, serving as timed progress markers. There are times when creating them is good. However, a more accurate timed progress marker is running tested code. It is more accurate because it is not a timed promise, it is a timed accomplishment. – Alistair Cockburn in Jim Highsmith, e-business application delivery, Vol. 12, No. 2, February 2000 [http://cutter.com/ead/ead0002.html]*

Tests are the best form of system documentation because they are the form that distracts least from producing releasable code. Don't settle for less.

How To Write Tests First

Before you write code, think about what it will do. Write a test that will use the methods you haven't written yet. Assume the methods exist and use them in the test. The test won't compile (if you have to compile things in your environment). Write the class to be tested, and its methods. Just a stub, not all the details. Your test should compile now.

Add the test to the test suite that holds all of your other tests for related stuff. Run the test suite. Your new test will cause it to fail. You don't have any implementation for the methods you're testing, so this shouldn't be a surprise. Running a test just to see it fail might seem a little strange, but it's important. Failure validates the test at this point. If it passes, it's clearly wrong.

Now write just enough code to get the test to pass when you run the test suite again. Doing things this way guarantees the smallest design that could possibly work. It keeps your code simple. Yes, there will be refactoring to do later, but it will be small.

Add your test suite to the suite of tests for the entire system and run it. That may seem redundant, but it isn't. A few days before OOPSLA '99, a project he was heavily involved with was coming to the end of an iteration. He was trying to add a new feature to the persistence framework. Pairing with someone who was new to the project, to Java, and to XP, Ken took charge. He coded like a madman for an entire day, writing tons of code without writing tests first (strike one) and not rerunning the test suite for the entire persistence framework (strike two). After struggling to get

the new feature to work, he rotated pairs and tried to integrate with a new partner. They ran the test suite for the persistence framework and got roughly 30 failures. Thinking out loud, Ken's pair said, "Let's see, they all ran with your changes before we integrated, right?" Ken felt like and idiot.

Save yourself this pain. Get addicted to running the tests. Made a change? Run the tests. Made a change that you didn't like and backed it out? Run the tests. Integrated? Run the tests. Took a break for coffee and just got back to your desk? Run the tests. You get the point.

Refactor the tests when they don't seem to be giving you the leverage you want.[2] If you notice that the suite of tests for a given class doesn't include something that should be tested, write the new test and run the suite to make sure it passes. If you need to refactor some portion of the code, and you aren't satisfied that the existing tests help you (maybe they don't cover everything well enough), refactor the tests first. Keeping your tests clean is just like keeping your code clean, only more important. Better tests give you more leverage to make things work better and to add to new features quickly.

In *XP Explored* (which should be published by the time you read this), Bill Wake gives a simple set of steps for what he called the "Test/Code Cycle" in XP:

- Write a single test.
- Compile it. It shouldn't compile, because you haven't written the implementation code it calls.
- Implement just enough code to get the test to compile.
- Run the test and see it fail.
- Implement just enough code to get the test to pass.
- Refactor for clarity and "once and only once."
- Repeat.

Bill claims this process should take about ten minutes per test. If it takes longer, start writing smaller tests. That may be a bit short, but it's not crazy. In early 2000, Ward Cunningham stopped by RoleModel Software, Inc. to check out how Ken's team was implementing XP. He paired with a number of folks on the team. Every person he paired with made one major observation: Ward took small steps, sometimes ridiculously small, but he moved like the wind.

Test-first programming is all about building confidence so that you can work at maximum speed. It only works if you test a lot, which means you have to take steps

[2] See *Refactoring* by Martin Fowler for the definitive reference on the practice of keeping your code clean.

small enough to force you to test often. Code a little, test a little. Get the tests to pass. Make a change. If the tests fail, the change must have caused it. If you write a few tests, then code for a couple of days, where is the error when a test fails? You'll be hunting for a while, which will slow you down. [WHAT'S A GOOD ANALOGY HERE? SOMETHING ABOUT PROBLEM BUILD-UP…]

What To Test

The rule of thumb we use is to write tests for non-trivial things that could break. We tend not to write tests for getter and setter methods. We've also learned to shy away from writing tests for methods that simply invoke another method, as long as that method already has a test. You'll come up with your own exceptional cases for when you don't need to write tests.

Err on the side of having too many tests. When in doubt, write one. If you need the confidence that your getters and setters absolutely will not break, write tests for them. There aren't hard and fast rules here. Write they tests *you* need.

Very few of us like writing tests, but we lave having them. It's some extra work, but it's well worth it. Until you have a bunch of tests and a lot of experience getting burnt by not having one you need, it's better to have too many.

How To Start Writing Tests First

You should be convinced that you need to write tests first. You might have no idea how to do it. Relax. You're in good company. Every time we talk with people who aren't used to writing tests first, we hear things like

- How exactly do I write a test first?
- How would you write a test first for XYZ?
- Huh?

To be honest, most people "in the know" about XP respond roughly the same way. They say they can show you how to write tests before you write code, but they can't describe it to you. This is a cop out. This is not a cop out. That was not a typo.

Asking someone how to write tests first is like asking someone how to write source code. You can't code-by-number. The code you write depends on a host of variables. Likewise, there isn't a comprehensive set of rules for you to follow when writing tests. The tests you write will depend on the code you want to build to exercise the tests.

Unit tests are just another type of source code. You write them, compile them if you have to, and run them. If you use xUnit[3] (and we recommend it), it's just like writing code that employs a small library of components, methods, or functions. The only real difference is the goal.

Old habits die hard. If you aren't used to writing tests first, it feels awkward. Take comfort in knowing that

- eventually, it will become as natural as writing source code
- your first tests will stink, but that's okay
- your tests will improve with practice
- even crummy tests are better than none

The key to getting into the habit of writing tests first is to realize that you've been doing it for years, but you probably didn't know it.

When you write code, you are imagining the tests. You just haven't written them down yet. If you're writing some code to compute a salary, you're thinking…get the base salary and bonuses for a given employee from the database…compute FICA… determine the employee's tax bracket…compute withholding…compute take-home pay…and so on.

Stop. You already have imagined how the code is supposed to work. The only thing you have to do now in order to write a test first is to write a test method for the first step that assumes that step is in the code already. Write a method that calls the as-yet-unwritten method getEmployeeSalaryData(String employeeName). Give it a known employeeName and check that the results are what you would expect. The test will fail, of course, because the method isn't there.

Write the real code for getEmployeeSalaryData(). Run the test again. Fix the code until it passes all the tests. That's all there is to it.

Some things are harder to test than others (see *Testing User Interfa*ces below). Sometimes it's hard to imagine the tests first. Keep at it. It will feel more natural in time.

Testing User Interfaces

Testing user interfaces is royal pain in the backside. We get asked a lot how we write tests for these things. We have lots of answers, but they all stink. The bottom line is that the coverage we get from our user interface unit tests seems significantly

[3] You can download an xUnit framework for writing tests in Java (JUnit), Smalltalk (SUnit), VB (VBUnit), C++ (CppUnit), and many other languages at http://www.xprogramming.com.

lower than what we get from testing our business logic. The value received from the tests doesn't seem to be worth the effort put into writing them.

The best way we've found to test user interfaces is to isolate what you're testing first. Separate the UI from the business logic behind it as much as possible. Test the business logic with unit tests, which make the UI less prone to break. Of course, that just means there is less to break, not that it's been tested adequately. Try to write unit tests for what remains in the UI (probably because it belongs there). You probably won't be able to test everything this way, but that's all right. It's better to have solid unit tests for 90% of the system than none at all.

One of the best ways we've seen to test user interfaces is to test as much as we can with unit tests, and then use functional tests to fill in the blanks.

Functional Tests

Functional testing ("acceptance testing" is probably a better term) is actually more important than unit testing in many ways.

There is no functional testing equivalent of JUnit. You can't just download something and start testing on day one. Even if you could, you still would need customers to help define the tests.

Customers often have as much trouble writing functional tests as developers have writing unit tests. Give them time. Figure out how you can automate these. Run them nightly. Give your customers the same kind of confidence boost that unit tests give developers. Without functional tests to prove a customer story "works", it doesn't exist. It certainly can't ship.

We use a functional testing framework affectionately known as JAccept™. The customer enter tests into Microsoft Word tables. The tests consist of user actions that can be performed on the UI being tested. A set of Java classes read the test files each night (we actually generate HTML from the Word files which the classes then parse) and run the tests automatically. Results are written to an HTML output file. We're currently working on porting the entire thing to XML, using Ant as the file dependency engine, and adding a customized XML editor on the front end.

The way you do functional testing isn't as important as doing it. Functional tests are the only thing that prove to your customer that the system "works." Without these, you're telling your customer to trust you. We hope they can, and we hope they believe they can. Proof removes all doubt. It is the customer's equivalent of the xUnit green bar. Nothing gives a programmer a shot in the arm quite like seeing that wonderful green bar. Give your customer the same confidence.

Functional tests also give the customer a real idea of how close the developers are to being “done”. This lets your customer make informed decisions about whether or not a system is ready for release.

Automate functional testing whenever you can. This makes it a normal part of life. Run them daily. This will give your customers the confidence they need to remain enthusiastically involved with the project.

[Need to insert some of our Acceptance Tests stories.

Chapter 14

Pair Programming: Stop the Maverick

Two heads are better then [sic] one.
-- John Heywood, Proverbes

Force people to break the habit of working alone. If you don't, you will not see maximum productivity.

Ever try writing with your non-dominant hand? Pair programming can feel like that if you're not used to it. It is one the hardest XP practices for developers (and managers) to adjust to. We've already mentioned the economic reasons why pairing is superior to solo development[1], but let's face it. Pairing seems a little weird. But it is absolutely essential to your success in XP.

Programming in pairs increases code quality. If code reviews increase quality (and they can), why not do them all the time? That's just one of the benefits of pairing. Nothing keeps you honest like having somebody sit next to you while you code.

Programming in pairs increases developer speed, which reduces development time. XP is structured to let customers get value out of their software sooner rather than later. Money today is worth more than money tomorrow. If you are working in pairs, you can stay on the road, without wasting time digging out of ditches.

Programming in pairs reduces project risk. If you're a manager, that's music to your ears. If you're a developer, you're happy when your manager is happy.

Code Quality

Code reviews are quite possibly the best way to improve code quality. Over ninety percent of the benefit from code reviews come from the first reviewer.[2] So, XP makes this the norm. We pair almost all the time, which means another person is reviewing our code constantly. There is no better way to ensure code quality.

---

[1] You might also want to take a look at "The Case for Collaborative Programming" by John T Nosek in *Communications of the ACM*, March 1998, Vol. 41, No. 3.
[2] No clue where this comes from, but it's a great stat.

Very few people like doing code reviews. You didn't get to participate in creating the stuff, in solving the problem. That's the exciting part. Reviewing code is sort of like proofreading someone's doctoral dissertation on the mating habits of fruit flies. Even if it weren't, it wouldn't get done well, if it got done at all.

While it is true that if you don't have time to do it right, you won't have time to do it over, on most projects there isn't time for either one. When Roy was a bit younger (and he is *not* old, mind you), he worked on a huge project with over fifty developers. The schedule was stupid. The team took beating after beating and had to come back for more. Roy had to schedule his bathroom breaks, for crying out loud. When, pray tell, was there room for code reviews?

Code reviews go by the board when the schedule's on the line. When they do get done at all, they're often cursory affairs conducted by people who are under such extreme stress that they aren't able to give it their best. Worse still, they probably don't know enough about the details of what they're reviewing to do a credible job of it. A crummy code review might be slightly better than none at all, but not much.

XP doesn't give you a choice. You have to review code if you are to engaged as a pair. You do it all the time. It's part of *writing* the code. While someone else is typing, you're reviewing their work. If they do something stupid, it's your job to point it out (politely, of course). It's their job to do the same for you when it's your turn to drive.

Code reviews after the fact are a fine way to get some of the benefits of pairing. In fact, they can help projects produce better stuff.[3] But they are a distant second to the quality improvements you get from pairing.

The Need for Speed

Football is a psychological game. No, really. If you watch enough of it, you'll hear commentators talk about momentum a great deal. The team that's making the big plays, getting the yards when they need them, marching down the field – they've got the "Big Mo." Despite talent and preparation, a momentum swing sometimes is enough to put a team over the top.

Momentum on a software project is about speed. If you're chugging through your iterations producing great value for your customer with each one, you've got momentum on your side. If something slows you down, you can lose the game despite the talent you have or the preparation you put in. You should guard your velocity with your life, because your velocity *is* your life.

Two things more than any others can slow you down:

[3] A host of research here.

- Crappy code
- Poor communication

If your code stinks, you'll move slower. Pair programming, as an almost constant code review, improves code quality. Improved quality means you don't have to spend as much time on clean-up, or on hacks to make something work, or on bug fixes. Those things take time. They sap your energy. They slow you down. They represent death by a thousand cuts for a project.

XP is a team sport. People must stop working on their own, and keeping problems to themselves. Some people may experience short bursts of hyperproductivity when they program solo, but it will slow the team down in the long run.

One other thing to keep in mind is that when people pair, they don't waste as much time. A blurb on Ward's Wiki puts it well:

> *Two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don't check e-mail messages or favorite Web pages; they don't waste each others time. Two people working together will have their shared time treated by others as more valuable. If I'm sitting at my computer, or just staring into space (thinking hard!), no one will think twice about interrupting me. On the other hand, if I'm busily working with someone, anyone who needs me will interrupt me briefly if at all. [http://c2.com/cgi/wiki?ProgrammingInPairs]*

Pairing keeps people honest. It keeps them focused. It keeps them moving at maximum speed.

Reducing Risk

Kent Beck says that if an approach to software development increases the probability that a project will stay alive to give you the big payoff at the end, it will be more valuable than the alternatives.[4] Pair programming is critical to helping XP do that.

Projects fail for lots of reasons, but there are a few primary things that increase the risk of failure, based on our experience:

- Slowness
- Poor communication

[4] *Extreme Programming Explained: Embrace Change*, Kent Beck, pp. 12-13.

- A high “truck number”

Pair programming minimizes these risks better than programming alone. We addressed slowness earlier in this chapter. We’ll hit poor communication in Chapter N. The last one probably needs some explanation.

Suppose the best programmer on your team gets run over by a bus. Now what do you do? Most projects we experienced before we found XP depended on heroes to keep them anywhere close to on track. If one of these folks went away, you could kiss the project goodbye. XP realizes that the only way to get past this is to reduce your team’s dependence on heroes. That’s what pairing does. It increases the number of people on the project that have to get run over by trucks (or quite) before the project is incapacitated. The common term for this is “truck number.” That was better than “carnage quotient,” we suppose.

There are times that we don’t pair and it might be the right thing to do. We just keep it rare in order to minimize risk.

How To Pair Program

Pair programming is

- two programmers
- actively working on the same problem
- on the same machine
- at the same time.

That’s some loaded language, bub. Let’s take each piece separately

There are two roles in pair programming: the driver, and the partner. Both are active. If you are programming in a pair, you must be engaged at all times.

When you’re driving, you’ve got to manage the mechanics of what you’re doing. “Should I use this method or that one?” You need to pay attention to what your partner is saying, or not saying. That person is there to help you, after all. Don’t look a gift horse in the mouth. If your partner make a suggestion or raises an objection, listen. If your partner isn’t engaged, slow down. Don’t just run off and leave him reeling. Ask him what he doesn’t get. Better still, let him take the wheel.

When you’re the partner, you’re not just riding and admiring the scenery. This isn’t “pair watching.” Think of your role as both navigator and partner. Actively understand everything that’s going on. Ask questions. Suggest alternatives. Stay on the same thought plane as the driver. Think beyond the code mechanics, to be sure,

but don't veer off on some crazy tangent. Don't distract the driver. Give him time to see his own mistakes and correct them. If he gets stuck, ask to drive.

If pairing is to be effective, people have to practice both roles often, and they can't be joined at the hip with a particular pair all the time. Switch places from driver to partner and back again with regularity. Switch pairs often, keeping one caveat in mind: don't switch pairs in the middle of a task. Ken says,

> *I find that I need to reach a critical mass with my pair: making sure we both know where we're starting from, what we're trying to accomplish, and the approach(es) we might take. I'd rather not do this too frequently. There are times that I've switched pairs two or three times on the same task. Try to avoid this. As long as I don't have to switch pairs in the middle of a task, I rarely am slowed down and almost always see my productivity increase.*

Before you can do any of this, you have to have an environment conducive to pairing. That's right, it's time to move some furniture.

When Roy was a bit younger (and he is *not* old, mind you), he worked on a huge project with over fifty developers. Everybody sat in cubes shared by two people. You couldn't directly collaborate with anyone but your cubemate. And you couldn't just switch places with somebody else…that was the unpardonable sin. You weren't allowed to remove walls entirely and reconfigure them either. This was silly. It also didn't work. The team ended up removing panels to makes "windows" in the cubes. They had to jump through too many hoops to do what should have been natural.

Rearrange the workspace to facilitate XP. Don't be shy. If people's cubes aren't next to each other, change that first. When they're all together, make sure each cube has enough space at one desk for two people to work side-by-side and still get both sets of legs under the desk. Then tear down the walls. You shouldn't have any walls between programmers on the same team.

On last thing. Once you're pairing, switching up as often as practically possible, don't let a bad apple spoil the whole barrel. If somebody is resistant to pairing, or simply refuses to do it, don't put up with it. We have found playful joking to be successful in getting people over their resistance. When somebody mavericks, we scream, "Witch!". That's a subtle way (or not so subtle when Ken screams really loud – he's got quite a set of pipes) to remind people of what they're not supposed to be doing. It usually does the trick. When it doesn't, the coach should talk to that person to explain the reasons behind the rules. If they still don't shape up, it's time to get a little tough. If they flat-out refuse to go along, bid them farewell. Here's how Ron Jeffries puts it:

> *What I do (as opposed to how I talk) is that when someone is transgressing rules in some way, I'll have a talk with them. Usually something bad will have happened, which lets us begin by agreeing that something bad shouldn't happen again. I relate the rules to the bad effect in an impersonal way. "The UnitTests weren't run before the code was released. The UnitTests would have shown the problem. That's why we insist on running them every time. We're fallible, the computer isn't." Usually the problem won't occur again. Also I watch the person and nag them a few times in a friendly way. Perhaps most importantly, I'd coach the other team members to watch out for the bad behavior when partnering. In other words, gang up on him. If it does happen again, I'll have a still friendly but serious chat that says "Everyone in the group really does have to follow the rules." Since they're programmers, they can do the math on people who don't follow the rules. If it happens a third time, I would politely but gently remove them from the group. If removing them isn't possible, I'd make it turn out that they didn't get to work on anything important. [http://c2.com/cgi/wiki?ProgrammingInPairs]*

Sometimes a little tough love goes a long way. In the end, though, XP isn't for everyone. Most people who try it love it, but some don't. Find these people and help them exit gracefully, for the health of the rest of the team.

Taking It To The Next Level

All this talk about how to pair is great, but the really wondrous thing about pairing is the interpersonal relationship part, not the mechanics. Anybody can sit next to someone else and throw in his two cents every so often. Many people can be completely engaged and try to make the result better. But the ones who really understand pairing know that it's about loving another person.

Yes, you heard right. Pairing is about loving the person you're pairing with. We're not talking about the romantic kind of love here. We're talking about the kind that seeks to show itself through actions. You will only get the most out of pairing if that's the attitude you have. Here's what we mean.

If you are loving another person, you are trying to see the best in them. You are trying to help them be the best they can be, rather than just looking out for number one. You are investing in their growth, and coming out better for the experience. You will be patient with them. You will be kind. You won't be jealous of their success, but will rejoice with them when they kick butt. You won't build yourself up at their expense. You'll take a back seat sometimes and let them shine. You won't hold a grudge when they screw up and it hurts you. Instead, you'll forgive them. You will support them when they're down and be on their side when the going gets

tough. As Bill Wake says in XP Explored, “Pair programming asks us to accept our humanity and continue.”[5] If you’re loving your pair, you’ll accept *his* humanity and continue.

Ron Jeffries was on to this when he said

> *Extreme Programming (and leadership in general) is a work of love. If you don't respect others, you're not doing it right. I try always to let my great respect show through for people who try hard to do the right thing. And sure enough, they do try, in almost every case. The others, who are perhaps trying in some way I don't understand ... I respect them too ... and wish them success elsewhere. [http://c2.com/cgi/wiki/?EnforcingMethods]*

How should that attitude affect how you pair? It’s taking pairing to the next level. You won’t just contribute. You’ll be humble and polite, truly respecting what the other person has to offer. You will talk so that your pair can keep up with you, rather than leaving him in the dust. [6] You will listen when your pair offers some wisdom or advice, because, hey, you don’t know everything. You will be gracious when your partner makes a mistake, gently giving him time to see it himself. You will let the other person drive sometimes, even if he’s junior. You will notice when your pair makes a great contribution, and be his biggest fan. You will respect each other enough to know your partner’s rhythm. You will speak in “we” terms, not “you” terms, unless you’re paying a compliment. You will speak in “I” terms when things get rough.

Radical? Sure. But that’s taking pairing to the next level. The question of why you should have this attitude in general is the subject of another book on XP (Roy’s working on right now, in fact). Suffice it to say here that it makes pairing a much richer experience than simply going through the motions.

The Inevitable Objections

In Chapter 3 we talked about resistance to XP. Most likely, the resistance you face will concentrate on pair programming. It’s an easy target for knee-jerk myopia. Fortunately, you have anecdotal and empirical evidence on your side. We think the best summary of both is in an article published by Laurie Williams of NC State University and Alistair Cockburn (pronounced “Coburn”, like James Coburn) of

---

[5] Page 94 in his draft, not sure what the reference will be in the final.

[6] Take a look at the *Conversant Pairing* paper by Nathaniel Talbott at http://www.pairprogramming.com for some excellent opinions on how to do this. It is a great elaboration on what Ward Cunningham calls .”reflective articulation.”

Humans and Technology. We'll summarize the key points here Laurie and Alistair deserve all the credit.[7]

Managers view pairing as "wasteful" in terms of time and money. Programmers resist it as well, because they aren't taught to do things this way and they feel it steals their individuality. Alistair and Laurie investigated these objections and summarized their findings this way:

1. **Pair programming saves money.** Pair programming increases development cost by about 15%, but the resulting code has 15% fewer defects. Based on a simple example they give, this can reduce "fixing" costs (in testing, in QA, or after release) by 15% to 60%.
2. **Programmers like it once they try it.** Consistently over 90% of students studied said they enjoyed their work more when pairing. Professionals surveyed on the internet said the same.
3. **Design quality improves.** Pairs produce higher-quality code, and they implement the same functionality in fewer lines of code than solo programmers do.
4. **Code quality improves.** We all know about the exponential cost curve. XP flattens it, but Laurie and Alistair looked at pairing in isolation. Pairs find mistakes sooner and follow coding standards better (and learn to talk and work together better). This improves code quality.
5. **Pairs solve problems better and faster.** Pair combine brainstorming and "pair relaying" (tag-teaming when one or the other gets stuck) to maximum problem-solving effectiveness.
6. **Pairs learn better and faster.** Pairing is basically revolving apprenticeship. The partners alternate learning and teaching from moment to moment. "Even unspoken skills and habits cross partners."
7. **Pair programming improves team building and communication.** Pairs communicate all the time. Rotating pairs increases information flow. This increases team effectiveness.
8. **Pair programming facilitates management.** Having people program in pairs increases programmer skills faster (they're learning all the time). Everybody is familiar with key parts of the system, so the project faces lower risk from losing key programmers.

That's some pretty good ammunition. If that fails, get some experiential evidence by giving it a try. Taking pot shots at something potentially beneficial without actually trying it is just being pigheaded.

[7] There are lots of places to find info about their research. You can find the "official" version (the actual paper they presented at XP2000 in Sardinia) at http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF. You also should visit http://www.pairprogramming.com. The site has oodles of information on pairing.

When Not To Pair

This may be heresy within the XP community, but there are certain rare times when it makes sense not to pair. Notice that important word “rare.” That’s more than a way to cook meat.

When not to pair will vary by team. Our rule is that mavericking shouldn’t happen often. It should never happen on tough code. But there are some instances when it makes sense not to pair. In these cases, working in parallel is smarter. Ken calls this Pairallelism™.

There are only two cases where we’ve found it best not to pair most of the time:

1. when exploring something new
2. when debugging

When you’re exploring, it’s wise to give both members of the pair room to move in their own direction at their own pace. Some of the more extreme XPers would say that any code developed solo should be thrown away and then rewritten with a pair. We say, give us a break. We’re prepared to refactor or rewrite it if necessary, but we aren’t going to do it just because we *might* benefit from doing it. Pair until you reach something you don’t understand. If you struggle with it together for a longish amount of time, split up and search alone. Keep each other aware of what you’re finding as you go. When you think you know enough to move on, pair up again.

When you can do it, don’t pair when you debug (debugging sessions should be rare, after all, if you have good tests). Trying to follow a line of thought when somebody else is clicking all over the place is frustrating. When you hit a rough spot, split up and track down the problem separately. When you think you have the problem on the run, come back together as a pair and share what you learned.

Both examples illustrate a fluid pairing environment. Not only will you be switching pairs, but you’ll be moving in and out of your current pair when it makes sense.

Some people suggest that a second set of eyes after you have done some solo development is just as good as pairing. If you’re serious about it, it can work well. It’s better than mavericking with nobody to keep you honest. The main problem we see is that when we don’t pair program, we’re more prone to assume we don’t need the second set of eyes. There is also something lost in the transfer of knowledge. The second pair of eyes learns what has been done, but didn’t influence or learn from the path to get there.

When in doubt, pair and see if it works. Odds are it will.

Personal Space

One of the arguments we've heard against pair programming is that people need their personal space. Someone on an XP mailing list somewhere said that depriving people of place to call their own dehumanizes them.

[insert Steve Hayes' contribution here???]

We don't know of any XP proponents (including ourselves) who want to deny people personal space. The point is that cubes are a lousy place to have a team working environment. You might be surprised to find out that XP teams often choose to give up personal space voluntarily, but if your team needs it, find a way to give them some. Just don't substitute it for a team working environment. Give everybody a drawer, or a locker, or a cubby.

There is a big difference between valuing individuals and promoting individualism. XP values the individual so much that it chooses not to isolate them. If isolation is "humanizing" then we should all find employers who prefers that their people telecommute, stay or become single, order everything over the internet, and don't participate in team sports. And never, ever leave their homes. We have found that people warm to idea of "team space." They start valuing the team environment more than their personal space and might even give it up, or they will spend lots of time in their personal space and avoid the team. If the latter happens, you need to fix it. Help such people realize that they'll either have to shape up or ship out, or help them find work somewhere else.

You can actually make team space more attractive and get more people per square foot. Here's a picture of the main room in our studio:

How To Start Pair Programming

Do it for half a day. Many people say that pairing is such an intense learning and doing activity that they can't do it for a whole day. Treat it like running. If you've been a couch potato for five year, you shouldn't try to run a marathon. Likewise, give yourself time to work up to full-day pairing sessions.

Switch pairs often, especially when you're starting, but not in the middle of a task. Let yourself get used to rhythm of switching, but stay with a pair long enough to get the full experience of melding minds with that person. The goal is to pair with lots of people to get used to splitting pairs and initiating new ones at natural points. Pair with people of different levels and personalities. As Chet, Ron, and Ann said,

> *"It takes a little time to get used to pair programming, and, at first, it will feel a bit awkward. Then you'll get good at pairing with someone, and finally you'll get good at pairing with most anyone. [Extreme Program Installed, p. 90]*

The point is that you shouldn't just stick with one person. If you do, you'll start believing that pairing with that person is the model for how all pairings should be. That's silly.

Be introspective about this. Be open to learning where you aren't good at pairing. Then work on correcting your own shortcomings. Attack your own resistance to pairing before you get on somebody else too much. The neat thing is that after you pair for a period of time, you'll find that you don't like coding any other way. In *Extreme Programming Explored*, Bill Wake quotes a seasoned programmer as saying, "I have found that, after working with a partner, if I go back to working alone, it is like part of my mind is gone. I find myself getting confused about things." That happens all the time. It's the running joke where Ken and Roy work.

When Roy first joined Ken's company, he was brand new to XP. He was convinced it was the correct approach, but he didn't have any experience doing it. In his first week, he felt like he wasn't being very helpful to his pairs at all (he didn't even know Java very well). Then one day, as he and Jeff were talking during a break, Jeff said, "I can't believe it. I worked on that problem for four hours and come up with nothing. Then you sat down and we had it fixed in a half-hour. That's cool!" Pairing truly is amazing.

One temptation people face when they're trying to get used to XP is that they don't want to jump in the pool. As soon as they find a particularly knotty problem in the pairing dynamic, they punt and start tweaking the process. Don't. Pairing isn't easy, and it is hard to get used to. Do it "by the book" first, then figure out what to tweak and how. If you never do it by the book, you won't be able to be very objective about what you might be missing.

Leftovers

Story about pairing at NationsBank

Pairing as negotiation?

[Nathaniel's convervational pairing…]

[Kevin Johnson's pairing experiences]

Chapter 15

Refactoring: Making It Right

In anything at all, perfection is finally attained, not when there is no longer anything to add, but when there is no longer anything to take away.

--Antoine de Saint-Exupery

> *If you do not develop the habit and discipline of refactoring your code, it eventually will be nearly impossible to move at top speed. This will happen sooner than you think.*

Remember the mantra Ken drums into the people on his teams? Make it run, make it right, make it fast. Refactoring is the technique of improving code without changing what it does. It is the "making it right" piece of the puzzle.

If your system is to last for any length of time, it is inevitable that your code will change. Denying it is futile. The question is how best to accommodate that reality.

Refactoring keeps you ready for change by keeping you comfortable with changing your code. You just remove unsightly buildup out of habit no matter where or when you find it (within reason). This keeps your code clean, which will let it survive longer.

Refactoring also makes change possible. It is an investment in future speed, and the future could be tomorrow. The cleaner your code is, the faster you will be able to move when it comes time to change existing features or to add new ones.

Refactoring has another advantage. XP says you should write the simplest code that could possibly work, but it also says you'll learn along the way. Refactoring lets you incorporate that learning into your code without breaking it.

Being Ready for Change

Someone said that thoughts breed actions, which breed habits, which breed character. You do not have to think about acting on a habit. You just do it. In the software world, we have been taught that change is dangerous. Change causes delay. Change costs too much. It is risky to change a system close to production, or in production. And so on. We have a habit of avoiding change. We must break it in order to move forward. The only way to do that is to form a new habit that takes the fear out of change.

Refactoring gets you in the habit of changing your code. Once we have code that runs, we don't waste time debating refactoring too much, unless it's a major refactoring. When we see an opportunity to do it, we do it. The only way to get over your fear of change is to face it and kill it. Refactoring is your weapon.

Making Change Possible

It may seem counterintuitive that you should invest more time now to save time later, but it is true throughout life, not just in programming. This is the principle of deferred gratification. It is investing time and effort into something without getting the big payoff right away.

Last year, Roy was a couch potato. As his weight began to balloon, he decided to start running. Like an idiot, he did too much too soon and hurt his knee. He expected to get in shape without working for it. After his injury, he wised up and began by walking. Pretty soon he was running. Within four months he was running five miles a day. Within a year he was running five miles in thirty-five minutes. That's deferred gratification. [BETTER EXAMPLE?]

There is no use ignoring that refactoring your code takes more time than writing it and leaving it alone. But this is true only at the time you write the code, and only for a little while.

If you refactor your code after you add a new feature, this can take a few minutes or a few days. You can avoid that cost in the short term by skipping that refactoring. Soon, though, the creeping crud will begin to overwhelm your code. Beck and Fowler said in *Refactoring*, "Accumulation of half-understood design decisions eventually chokes a program as a water weed chokes a canal."[1]

Without a constant cleaning process, clutter will build up. That obscures your design, which makes it harder to preserve. That makes the XP practice of Simple Design impossible. Refactoring is the essential first step toward maintaining the simples possible design. Clean code simply gives you more room to maneuver. It is more flexible. Michael Feathers put it this way on Ward's Wiki:

> *I used to think that systems could be made more flexible by adding things. Now I realize systems become more flexible when you take things away. The area around the code has much more potential than the code will ever have. [http://c2.com/cgi/wiki?OaooBalancesYagni]*

[1] Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999), p. 360.

Adding stuff is much easier than removing it. Refactoring keeps your code as simple as possible, so that you can focus on adding things. When you boil it down, the primary of goal of refactoring is to keep your code easy to modify so that you can move at maximum speed indefinitely.

You will get some immediate gratification from even a simple refactoring. But you will see the real benefits of it in the long run. You have to trust that you will see these rewards. When you do, you can't go back to being afraid of change. It costs too much.

Putting Learning Into Your Code

The simplest thing that could possibly work changes over time. The simplest thing that could possibly have worked yesterday won't cut it today. It will change again tomorrow. If you are paying attention, you are learning along the way.

Refactoring lets you build that in. It helps you form the habit of changing your code so that it doesn't scare you. It keeps your design as simple as possible so that you can see where new things should fit. It helps you understand your code.

Fowler describes an interesting phenomenon in his book on refactoring. He talks about using refactoring to help him understand code. In fact, one of first things he does when he meets new code is to think about refactoring it to make it more understandable to him. That is building learning in. Something that was (perhaps) clear as crystal yesterday ended up being Greek today. Refactor it to make the intent clear.

Refactoring is the practice of refining your code as you get smarter. A nice side-effect is that being temporarily dumb ("What in world was I thinking here?") won't cost you anything. Refactoring helps you keep your code so clear that the code can answer your questions when you temporarily forget how brilliant you were.

How To Refactor

Martin Fowler wrote a book on refactoring that you should buy, read, and put into practice. Here are what we see as main "how" points[2]:

- Develop the habit of writing tests first and running them compulsively before you even think about refactoring.
- Write your code first, then refactor it.
- Treat adding a new function as an opportunity to refactor the code around it.

[2] Most of the material here is from Chapter 2 of *Refactoring: Improving the Design of Existing Code*, specifically pages 57 and 58.

- Treat fixing a bug as a refactoring opportunity, too.

Refactoring is a way to improve code without breaking it. The only way you can prove you didn't break the code is by testing it every time you make changes. Refactoring will work is if you have units tests to back you up. In fact, testing is such an integral part of refactoring that we can't imagine refactoring without having tests to confirm that we didn't break anything. Bugs in software are like a big game of Whack-A-Mole. You fix one and another one pops up. The same one probably pops up over and over again. You tire yourself out (and probably look stupid) flailing about. Running *all* of your tests often helps kill this problem.

So, remember to write the tests and the code first. Then refactor both to make them "right" (more right is probably better). This will improve your speed later in two ways

1. **It will make changes less risky.** Refactored code is easier to change without introducing bugs that slow you down. It also will make it easier to find new bugs that crop up, because your code will be clear.
2. **It will make optimization easier.** You eventually get to making it fast. Refactored code is easier to optimize because you can pinpoint performance bottlenecks easier, and you can make the changes to correct them in one place.

Refactor whenever you add something new. If you put on your refactoring hat when you hit code you need to modify, you will come to understand it better by making it easier to add the new feature.

The same holds true for fixing bugs. Refactor the code to improve your understanding of it while you're trying to figure out what the problem is. This alone can help you find the bug. If it doesn't, at least you'll leave the code better than how you found it.

When To Refactor

Kent Beck came up with the idea of "code smells" to describe that uneasy feeling you get that should tell you when to refactor. When you catch a whiff of one of these stink bombs, you should clean it up. If you think it's ripe now, let it sit for another month. You won't be able to get near it without protective clothing.

To be honest, though, some people aren't as sensitive to smells as others are. They need a little help. Fortunately, there are two regular opportunities for every programmer to be extra sensitive to smells: before implementing a feature and after implementing it. Developers try to determine if changing existing code would make

implementing the new feature easier. Developers look at the code they just wrote to see if there is any way to simplify it. For example, if they see an opportunity for abstraction, they refactor to remove duplicate code from concrete implementations.

The point is not to procrastinate. Refactor when you see the opportunity. Do it as you go along. See something in the code that takes you a while to figure out? Refactor it so it is clear immediately upon looking at it. Fowler says such refactorings help him understand the code better, so he does it whenever he sees new code (assuming it needs refactoring, that is).

When Not To Refactor

Fowler says that you should not refactor when the code doesn't work and needs to be rewritten.[3] That's where his list of times not to refactor ends. We agree. Refactoring needs to be as much a part of writing code as writing code. In fact, the reasons people give for not refactoring aren't good reasons at all. They are excuses for laziness or justifications for fear.

The more important question is one of degree. How much refactoring is enough?

When To Stop Refactoring

The knee-jerk reaction of most people who think refactoring is good idea is, "Never!" We understand the sentiment, but we disagree a little bit.

One of Roy's managers used to say, "Good is better than perfect." Sometimes that's true. As admirable as it is to try to get all the gunk out of your code, it's unprofessional not to ship a system because there is room for improvement. Being professional doesn't mean being perfect.

Every programmer (us included) finds himself leaving code in the system that he thinks could be better. The key is never to insert smelly code that has a known cure, and to remove the existing smelly code the next time you have to add something where it lives. Never just pass by smelly code and say, "It stinks, but I'll get to it later." Later won't come soon enough to save you from drowning in your deferred gunk. Ken describes it this way:

> *I find that Java's typing model causes me to encapsulate casting in order to keep users of a class from having to cast. After coding on a system for a while, I often end up with lots of small classes that exist mostly to encapsulate the generic stuff underneath, and to insert some type of intelligence. The code looks almost exactly the same as a couple*

[3] Fowler, M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley (1999), p. 66.

other classes that also add similar intelligence for a different type. I hate it, but it's not always obvious how to get rid of it.

Sometimes, I learn a new trick that makes it obvious. I usually don't immediately find all of the places in the system where I can apply that trick. The customer would be rather bothered if I took two or three days to do that and missed the iteration. However, I do communicate the trick and encourage people to apply it when they're in the code containing the particular suspect cruft. Eventually, the cruft works itself out and new flavors of cruft work their way in.

Don't refactor beyond what the customer wants. You'll get diminishing marginal utility. There will be some refactorings that you won't be able to get done, just like there are some features that you won't be able to include. Ward Cunningham describes unfinished refactorings as going into debt on a project. The future reductions in project speed are the interest you pay on that debt. Sometimes, a little debt is good management. You just need to manage debt well by paying it off with refactoring.

How To Start Refactoring

Start by reading Martin's book. Get familiar with the code smells and the appropriate refactorings to sanitize them.

Then start with what you know. Refactor before you add a new feature, and after you finish adding it. That will get you used to the process. At the same time, practice looking for specific "smells" that Beck described in *Refactoring*. Over time, your olfactory sense will get better at picking up bad code odors. If you've been around garbage all your life, you might not notice the smell. But if you force yourself to recognize garbage when you see it, it won't be long before it starts to smell bad to you.

Don't refactor too soon. Get your code to run first. Hack a path through the jungle, then figure out if there is a better way to go. Make sure you have a solid test suite (ideally, both unit and functional tests). Then start refactoring.

The real trick is to develop the habit of refactoring. Not all of your refactorings will make the world safe for democracy. Some are pretty basic. But the habit is priceless.

Why People Don't Refactor

If refactoring is such a great thing, why doesn't everybody do it instead of letting crap build up in their code? There are three primary reasons: impatience, cost, and fear.

You see some short-term benefits when you refactor, but you get the big payoff later when you continue to move at maximum speed indefinitely. Deferring gratification takes discipline, but it pays off when you get to eat the chocolate cake.

The economics aren't clear yet, but the main driver behind the traditional cost curve is code that is difficult to change. Refactoring is one of the tools XP uses to flatten the curve.

Fear usually comes from ignorance. Habitual action overcomes fear. The United States military trains based on this principle. Basic training is essentially a gradual conditioning to fear. Recruits learn to obey orders out of habit, even if those orders seem silly at the time. This gets drilled into them twenty-four hours a day. This is to ensure that they will act instinctively in the heat of battle, when an untrained person might do the natural thing and run away screaming. The only way to get rid of your fear of changing code is to do it over and over again.

Chapter 16

Continuous Integration

asdf.
--

> *Not integrating your code more than once per day is a recipe for headache and decreased speed.*

"Continuous integration" doesn't mean "continuous" integration. As Kent Beck says, this is a slogan, not a description. XP does not say that you should integrate every second. It simply points out that even daily integration isn't enough to avoid integration nightmares. "Integrating More Often Than Daily" isn't nearly as catchy a slogan.

We integrate new code into the existing code base multiple times every day. If you don't do this, your speed will suffer. You will spend days, or even weeks, trying to fix the backlog of bugs surfaced during your big bang integrations.

Integrating this often also reduces your risk of missing dates. It spreads the risk of a single integration event around to multiple small ones.

Continuous integration also makes it easier to pinpoint the source of a problem in one of your small integration events.

Maintaining Speed

Both Ken and Roy have worked on projects that integrated in big bang fashion. Roy can remember one project where they had "integration meetings" to resolve conflicts, take ownership of bug fixes, and so on. These meetings lasted for hours.

Going for two weeks, or a month, before you integrate is working under one of two faulty assumptions:

1. Your code won't cause anybody else to blow up
2. Nobody else's code will cause you to blow up

Both are dead wrong. You don't understand the entire system so well that you can inoculate your own code against problems caused by somebody else. Problems

will occur. Ignoring that is delaying the inevitable and storing up pain. If you do that, you'll be up against time pressure that will make maintaining discipline tough. It will hurt badly, too.

Consider how continuous integration changes the picture. You code for an hour and then you integrate the new stuff. You don't stockpile problems. This lets you solve problems when they are still small, while you have time left to fix them. That lets you maintain maximum speed all the time.

Reducing Risk

Cecil B. DeMille created and directed some of the grandest epic films ever to come out of Hollywood. There is a scene in *The Ten Commandments* that shows the exodus of the Isrealites from Egypt. It involved a huge number of extras that was costing the studio a ridiculous sum. There was only one chance to get it right. To be absolutely safe, DeMille had three cameras set up to film the scene from different angles. If one failed, there would be two backups. It was inconceivable that all three would fail.

DeMille called, "Action!" and the scene was off and running. It went just like he'd planned. Everyone performed beautifully. When it was done, he discovered the first camera had failed. No problem, he thought, that's why we have the backups. He radioed to the second camera, which had been down in the crowd of extras, and was told that somebody had kicked out its electrical connection. By then he was feeling quite thankful for having that second backup. That crew was closer to him, so he called to them on a megaphone to see how it had gone with them. They called back, "Anytime you're ready, Mr. DeMille!"

You create crazy "only one shot" situations like that for yourself when you try to integrate lots of code in one big bang. The longer you wait before you bring things together, the worse off you'll be. The antidote is continuous integration.

We integrate multiple times every day. This is like distributing a force. It's the difference between standing out in the rain for a few hours and standing under Niagara Falls. If all the force is concentrated in a single periodic integration event, it will crush you. It will take you a relatively long time to recover, to sort everything out. By that time, you probably have missed an important date. Since the fixing effort carries over into your next phase, where you'll do yet another big bang integration, you'll probably miss your next date. Continuous integration says that it's better to fail fast and early.

This kind of perpetual lateness and error stockpiling is what makes many software projects fail. Integrate as often as you can in order to avoid that.

Pinpointing Integration Problems

Distributing the force of integration over multiple small integration events makes it easier to figure out what caused a problem when one crops up. Suppose you code for an hour and you get all your unit tests to pass. Then you integrate. You know that the last pair to integrate made sure all the tests passed in the integration workspace. If you get a unit test failure when you integrate, most of the time it's your new stuff that must have caused the problem. This is much easier, and it helps you maintain your speed.

How To Integrate Continuously

Each pair writes code at their pairing station. They write their tests first, then they write just enough code to get those tests to pass. After they take any significant step, they make sure all of the tests for the entire system run in their own workspace. The hard and fast rule is that nobody can integrate broken code. Then they move to the integration machine.

We have found that the single biggest aid to making continuous integration work is having an "integration machine". This is a separate computer, within earshot of the entire team, where each pair goes physically to integrate their code. Kent Beck talked about having a "refactoring hat" and an "adding code" hat. You also have a coding hat and integrating hat. Having a separate integration machine makes it clear which one you have on. It also lets the rest of the team know when they can integrate. The can integrate only when the integration machine is free.

When a pair moves to the integration machine, they run the tests. Yes, the last team to integrate was supposed to leave all the tests in working order, but we always want to be sure. The pair then updates the integration workspace for their project (i.e. they bring in any code that they changed). Then they run the tests again. If nothing breaks, they yell, "Fore!" to tell the rest of the team that they just integrated successfully. If something breaks, they work on it until all the tests run. That leaves the integration workspace "clean" for the next pair to integrate.

When the pair gets back to their own workspace, they bring in the newly integrated code.

If you hear more than one or two shouts of "Fore!", you know that you need to integrate right away. Working on old code that hasn't been integrated is a recipe for a headache the next time you integrate. So, keep up.

How To Start Integrating Continuously

Continuous integration is a habit, just like refactoring is. You just have to start doing it. If you aren't used to doing it, it might seem crazy. The reason it seems crazy to many people is that they are conditioned to fear integration.

If you come from an environment that doesn't integrate continuously, you probably view integration like going to the dentist. You know you should have been brushing and flossing more regularly, but you didn't. It's embarrassing. Now you have to endure hours under the dreading "pick" thing that they use to clean your teeth. You might even have a cavity. It's your own fault, but you have become conditioned to loathe visiting the dentist.

What's the solution? Simple. Brush and floss. This is one of those things that nobody particularly enjoys doing. You do it to avoid the pain that you will have to go through if you don't. Integrating continuously is the same way. Pretend that integrating this way will save you pain. Once you do it for a while, you'll have firsthand experience to prove it.

Techniques to Make It Easier

[we should probably add something about our versioning name/numbers which helps integration go quickly and identify who can help you when problems occur]

Section Three: The Rest of the Story

Once you have the XP essentials down, you should begin thinking about the rest of the practices. At that point, they make a difference.

These things aren't less important, they just have to be concentrated on less when you start. Some of them will naturally fall into place (Simple Design, Collective Code Ownership). Others might just take some time until there essential in certain environments (). If you are doing these within the first three days, great! If you’re not, that’s fine. These things are important, but you can defer "getting them down" until you have the rest of the practices humming. In a way, it's like iteration two.

Chapter 17

Simple Design

Out of intense complexities intense simplicities emerge.
–Winston Churchill

> *Change is inevitable. Keeping your design as simple as possible prepares you for change.*

Detractors claim that XP neglects design. On the contrary, XP says design should not be done all at once, up front, under the delusion that things won't change.

We're sure someone is thinking, "If XP promoted design, this chapter would not be left to the "Rest of the story, less-essentials" section.

Let's set the record straight. No one (or at least no one who has a clue) thinks that XP is a substitute for using good design techniques. XP and "good design" are basically orthogonal. Good designers will produce better designs in an XP environment. Novice designers will have more opportunity to learn about good design in an XP environment. When asked to pick a team, both of us will take a bunch of good designers over people who don't think about design every time. Then, we'll work in an XP environment

Big Design Up Front approaches work under the fallacy that you can look at a static picture of the horizon, stay still, and draw a perfect picture of how to get to the point you're looking at. XP recognizes that you are better off with a simple plan that will get you moving in the right direction, and that the horizon will change as you move. XP considers design so important that it should be a constant affair. We always try to have the simplest design that could possibly work at any point, changing it as we go to reflect emerging reality.

Defining Simplicity

What is the simplest design that could possibly work? Beck defined it as the design that

- runs all the tests
- contains no duplicate code
- states the programmers' intent for all code clearly

- contains the fewest possible classes and methods

Requiring a simple design doesn't imply that all designs will be small, or that they will be trivial. They just have to be as simple as possible and still work. Don't include "extra" features that aren't being used. Don't add unnecessary complexity.

Your design should be simple enough so that its intent is clear to the people who will be modifying it. That doesn't mean that somebody without any domain knowledge or history with the team should be able to pick everything up in an afternoon. But the folks on the team for a while should definitely understand it, and new team members shouldn't have to climb Mt. Everest to get up to speed.

Why People Don't Keep It Simple

Nobody comes out and says, "Our goal is to create the most complex design possible." The problem is that people miss the forest for the trees. They start with grand intentions of keeping their design clean. Then they get distracted. They forget that simpler is usually better. It's easier to understand and easier to change later. Why does this amnesia happen?

Software developers favor complexity out of habit. In school they learn that complex problems probably require complex solutions. Or they don't have the discipline to think about the problem first and look for the simplest solution. Complexity tends to happen naturally, which is why design tends to degrade over time. Fowler described this in making his case for refactoring as a critical software development practice:

> *The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. [Refactoring, p. 55]*

Even if that weren't true, developers are bad at predicting what they'll need later. They can't guess requirements very well. If they guess right about something they'll need later, it probably will look at least a little different by the time they get there. That's why you'll hear "YAGNI" a lot in XP circles. XPers know that most guessing ends up being wrong, and that keeping simple is the best way to avoid wrong guessing.

As if that weren't enough, if developers break the complexity habit and the internalize YAGNI, they don't refactor like they should. If you don't, you can't keep your design simple. This is because you "lose" the design in all the crap. If there's junk in the way, you can't see how everything relates, and where new stuff should go.

Keeping a design simple requires several things:

- the ability to think simply in the first place
- the courage to understand code and to change it
- the discipline to refactor regularly in order to keep entropy from destroying your code

These are skills that don't develop by accident. They require practice. Often, a programmer's background, habits, biases, and peers fight against development of these skills.

Why Do It At All?

Change is inevitable. Habits and skills that make responding to change easier dramatically increase a project's chances of success. Simple systems are easier to change. Keeping them simple makes change easier. Therefore, fight for simplicity as if your life depended on it. It will, eventually.

> *The compiler doesn't care whether the code is ugly or clean. But when we change the system, there is a human involved, and humans do care. A poorly designed system is hard to change. Hard because it is hard to figure out where the changes are needed. If it is hard to figure out what to change, there is a strong chance that the programmer will make a mistake and introduce bugs…. The program may not be broken, but it does hurt. It is making your life more difficult because you find it hard to make the changes your users want…. Any fool can write code that a computer can understand. Good programmers write code that humans can understand. [Fowler, Refactoring, p. 6, 7, 15].*

Fowler makes the case that this is where refactoring comes in. But refactoring is simply the primary means to an end. Human-understand simplicity, not technical elegance, is the goal.

How To Start Doing Simple Design

As we said before, simplicity is difficult to maintain. It's also hard to learn. The only way to start is to be a little ridiculous.

When you study economics in school, what do the examples look like? They are full of ridiculous simplifying assumptions that make the examples easier to understand. As you learn, you can get rid the simplifying assumptions and get closer to complex reality. It is the same with simple design.

Remember the rules from Chapter 8. Think simply when you starting focusing on keeping your design simple. Recall Roy's first day at Ken's company. Roy spent four hours on the problem and came up with something ridiculously complex. Ken came over to help and had a great solution with about one-third the code within thirty minutes. What was the difference? Roy overhead another member of the team ask Ken how Roy had performed on his first test. Ken replied, "He just needs to learn to think more simply." Exactly.

For your first cut at a problem, write something that is so simple it makes you laugh. Assume the simple way will work until you see that it doesn't. You'll be surprised at how close the ridiculously simple design can get you to solving the problem.

Simplicity is no excuse for not thinking.

Why It's Not Essential in the Beginning

Simple design is essential to making a project successful in the long run, but you need to be focused on developing a few habits first:

- thinking as simply as you can (assume simplicity will work)
- writing code courageously
- refactoring that code with a passion

Keep it as simple as you can when you start, but focus on developing these habits in the beginning. As Fowler said in his paper *Is Design Dead?*,

> *The best advice I heard on all this came from Uncle Bob (Robert Martin). His advice was not to get too hung up about what the simplest design is. After all you can, should, and will refactor it later. In the end the willingness to refactor is much more important than knowing what the simplest thing is right away.*
> *[http://www.martinfowler.com/articles/designDead.html]*

Simple design is not an accident. But once you have the core habits of thinking simply, forging ahead without fear, and refactoring, simple design will happen, and you will get better at it over time.

The Essential Design Tool

CRC cards are fine. We like them. But the best design tool is right here:

A white board!

Get away from the computer to do "design". It's too constraining. When you are caught up in the details, get away from the code long enough to see the big picture.

You should have an environment that invites this. We have whiteboards on virtually every wall in our studio, and they are always being used. It's amazing how people work without them.

Make sure they are available within every few steps any place people who are creating something work on the thing they are creating.

We were appalled at one client when we found there seemed to be 3 or 4 whiteboards in a 30,000 square foot facility (need to verify the size). We solved this problem when we moved into our new space by buying a bunch of mylar 4' x 8' panels.

Don't fight about whether XP focuses enough on design. Fight with your management as to whether you have enough white boards available!

Chapter 18

Collective Code Ownership

asdf.
--

> *Turf battles over code stand in the way of developing a system at maximum speed. Get past this by making each programmer responsible for the whole thing. No code is off-limits. Nobody likes to make a mess they'll eventually have to clean up.*

Any person on the team has both the authority and the obligation to change any part of the code to improve it.

Everybody owns all the code, meaning everybody is responsible for it. This allows people to make necessary changes to a piece of code without going through the bottleneck of the individual code owner. Making everyone responsible negates the chaos that erupts when nobody owns the code.

Collective code ownership isn't the norm in software development. Developers tend to resist it, usually for reasons like this:

- They don't understand what it means, so they assume it's like no code ownership at all
- They fear giving up control of "their" code

What Collective Ownership Means

Collective code ownership is not no code ownership. Beck talked about this in *Extreme Programming Explained*.[1] In the bad old days, nobody any of the code. Everybody changed code willy-nilly to suite his purposes, whether or not the change fit well with what was already there. Chaos was the natural result, especially with the dynamic nature of objects. Code grew fast, but it was a mess. In self-defense, development organizations opted for individual code ownership. The only person who could change code was the owner. This created stability, but change was agonizingly slow. If the code owner left the team, change stopped.

[1] *Explained*, p. 59.

No code ownership and individual code ownership don't work.

Saying that everybody owns all the code isn't the same as saying that nobody owns it. When nobody owns code, people can wreak havoc anywhere they want and bear no responsibility. XP says, "You break it, you fix it." We have unit tests that must run all the time. If you break something, it's your responsibility to fix it. This requires discipline. If you develop that discipline, collective ownership can work.

Moving From "Me" To "We"

Even if developers believe that argument, though, they still might resist. The biggest barrier to collective code ownership is a programmer that feels hurt by the practice.

We have seen programmers like this, and we can empathize. If you are used to owning code, it can seem intimidating not to own it anymore. It can feel like you are losing control, like you aren't responsible for producing anything. That can be scary. It also can be a blow to your ego. If everybody owns all of the code, you can't be a hero in the same way that you used to.

The only way to get past this is to conquer your fear and to beat your ego into submission. Neither one is easy.

Collective code ownership doesn't work without trust. You have to give up control. You have to give up the idea of exclusive ownership of "your" code. You have to believe that "we" produces vastly better results than "me". If everybody owns all the code, anybody can change any of it to be better (meaning, to meet requirements better). This will keep your code clean, and will increase your speed. If everybody owns all the code, your ability to deliver isn't destroyed when the owner of a particular piece of code leaves the team. Everybody owns that code. The only way to see this is to give it a good try. Get over the initial speed bumps and see if it will work.

The harder hurdle to clear is the ego problem. We talked about humility before. Nowhere is it more important that right here. You have to change your definition of what it means to be a hero. Heroes aren't the ones who have the best code with the most clever tricks – those are selfish glory seekers. XP heroes are the ones who make the team better – the best pairs, the best refactorers, the best encouragers of other people. If people don't change their thinking, it will be tough for them to change their behavior, no matter how compelling the argument is about results.

Why Have Collective Code Ownership?

Consider the alternatives to collective code ownership.

If nobody owns the code, nobody can be held responsible for messing it up. Chaos will follow, be sure about it. This is how it used to be. When there is a problem with integration, or a production bug, it will get ugly. Things will degenerate into finger-pointing sessions where the problem somehow isn't anybody's fault.

If individuals own parts of the code, it's not much better. In this model, code gets further and further away from the team's understanding of what the customer wants. You essentially have mavericks, even if they're pairing. They are always thinking about "their" code, worrying about how others are messing it up. They might even cheat and clean it up when their pair isn't around. Even worse, change ends up being practically impossible. It isn't worth the pain of going through submitting a change request that the developer will get to when he finds it convenient.

Either alternative guarantees that a development project will move slower than optimal speed. Collective ownership of all the code binds the team together. Everybody has a sense of ownership and empowerment. Turf battles go away, and the team can concentrate of the real goal: making the system users want and developers can be proud of.

How To Start Having Collective Code Ownership

There are three things your can do to start implementing the practice:

1. Check your ego at the door. Just give it up for a while as an experiment.
2. Look past the short-term discomfort of giving up control of code to the long-term benefits it can produce
3. Use tools that make mistakes easy to recover from and that provide robust version control

Giving up your ego isn't easy, but try it. Remember, it's for science.

Then act like you own all the code. Focus on the results the team can create if everybody acts that way. When somebody doesn't act that way, give them a hard time. The grand experiment will fail if even one person doesn't play by the rules.

Of course, a safety net can be a good thing. It's easier to experiment if the risk is low. It's hard to go fast when you're afraid. Good tools can be your safety net.

We use and Envy-backed tool for all of our Java development (IBM VisualAge for Java). An Envy repository is a wonderful thing. We have version history for every single change we make. We can always go back to a version that worked. Everybody on the team can keep themselves in synch all the time. If somebody

hammers something beyond all recognition, we can revert to a clean state. Without a tool like this, collective code ownership is harder.

Why It's Not Essential in the Beginning

As soon as you start breaking down work into stories and tasks, work on them in pairs, integrate continuously, and keep all the tests working, you will realize that you mostly have collective code ownership. If you don't, figure out why not. It's probably because you are doing something else wrong.

Talk about it at your Stand-up Meeting. (See why it's more important to do Stand-up Meetings than Collective Code Ownership?).

Chapter 19

Where's The Customer?

If the client won't give you full-time, top-flight members, beg off the project. The client isn't serious.

-- Tom Peters, The Professional Service Firm 50

> *Customers decide which system features get built first. When a question arises about a requirement, they have to be available to ask. The best way to ensure they are available is to have them be part of the development team.*

Over half of all software projects fail. Many of these failures are due to poor communication between the customer and the development team. The development team made assumptions about what the customer wanted or needed, and built the wrong system.

Having real users around to answer questions, to set priorities, and nip misunderstandings in the bud avoids this problem.

Why Have An On-Site Customer?

To function optimally, an XP team needs to have a customer available on-site to clarify stories and to make critical business decisions. Developers aren't allowed to do that alone. Having a customer available eliminates bottlenecks that can arise when developers have to wait for decisions.

XP doesn't pretend that a story card (or a requirements document) is all the direction a developers needs to deliver the necessary code. Each story is a commitment to a later conversation between the customer and the developer to flesh out the details. The idea is that communicating face to face minimizes the changes of misunderstanding, unlike writing all the requirements down in a static document.

Beck says that the process of defining requirements is a dialog, not a document. Having the customer around to clear up ambiguities and to provide direction keeps things on track.

The most common objection to having an on-site customer as a member of the team is that it costs too much. Real users are too valuable to give up their time. That is short-sighted thinking. Having real users around to answer questions for the

development team will ensure that the software will be working sooner. If that's not worth the output of a user or two, the system isn't worth the money.[1]

On-Site Versus Always Available

The customer needs to be available at a moment's notice to answer questions that will let the project move at maximum speed. If you have to wait a week to get direction, you're sunk. That said, it is important to be practical.

Sometimes, the customer cannot be located physically with the project team. That would be ideal, but sometimes it just isn't practical. We have found that a remote customer who is available at any time is good enough. We developed a framework for a wireless application company based across the country from our office. The customer was always a phone call away. If he was too busy to respond right away, he got back to us the same day. Very rarely did we have to wait more than a day for direction. It worked fine.

Similarly, sometimes a customer cannot devote every hour of every day to the project. While it might be ideal to have the customer sitting available at any minute for questions, that isn't always practical either. The important thing is that the customer has to be available when needed, within reason. If the customer were to sit with the team all the time, he would be wasting much of it. You simply don't have to talk to your customer every minute. We sometimes don't have to talk to our customers for several days. Having them sit there twiddling their thumbs isn't the best use of their time.

If you can't have an on-site customer, make sure you can have one that is available whenever needed. We wouldn't turn down work where our customer couldn't be physically present with our team. We *would* turn down work where our customer didn't care enough about the project to give the team whatever they needed.

How To Start Having An On-Site Customer

The best way to get an on-site customer for your team is to ask for one. If the person authorizing and paying for the project objects, confront that person with the choice of getting better software sooner or increasing project risk. They might cave.

If you can't get an on-site customer, do your best with an "always available" customer. That might work well enough. If it doesn't, document the snags and make the problem very obvious. Be a squeaky wheel. Don't accept failure as natural. Continue to press for an on-site customer if the alternative doesn't work.

---

[1] *Explained,* p. 61.

If you can't get either an on-site user or an always-available one, you have a simple choice. You can go ahead with the project and accept the dismal odds of success, or you can call it quits. Regardless of which option you choose, you should go in with your eyes open.

Why An On-Site Customer Isn't Essential

You can start doing XP without an on-site customer. Beg, borrow, and steal time from your customer to get going. Just recognize that you'll go a little slower. Once you're used to the practices, adding an on-site customer will help you make the quantum leap to the next level.

Chapter 20

Coding Standard

asdf.
--

> *Coding standards keep the team from getting distracted by useless trifles.*

Coding standards used to be a big deal. Some people claim that it is impossible to develop maintainable code without them. We certainly have seen plenty of cowboy code with formatting that could be described as “random”, at best.

XP says that having a coding standard is important. The particular one you pick really doesn’t matter that much.

Keep your standard simple and practical. Complicated standards get ignored most of the time.

Why Have Coding Standards?

Having a coding standard does two things:

1. it keeps the team from being distracted by stupid arguments about things that don’t matter as much as going a maximum speed
2. it supports the other practices

As Beck put it on Ward’s Wiki, XP requires you to spend so much time looking at code that somebody else typed that consistent formatting is quite important:

> *The conventions for naming and method size are more important than where you put returns and tabs, but knowing exactly where the white space will be when you look at a method reduces friction considerably. [http://www.c2.com/cgi/wiki?CodingConventions]*

Without coding standards, it is harder to refactor code, harder to switch pairs as often as you should, and harder to go fast. The goal should be that no one on the team can recognize who wrote which piece of code. Agree on a standard as a team,

then stick to it. The goal isn't to have an exhaustive list of rules, but to provide guidelines that will make sure your code communicates clearly. As Fowler said in *Refactoring*, the compiler doesn't care whether code is ugly or clean, but humans are the ones who change the system and they do care.[1] The more hoops you make people jump through to change the system, the more they will avoid change.

How To Start Having Coding Standards

If you have a current standard, use it. If you don't, code for a while to see if consensus emerges. Then implement a standard. Don't spend too much time up front trying to decide all of the coding rules. That will just slow you down.

Why Coding Standards Aren't Essential

For coding standards to matter, you have to be coding first. It is far more important to be making progress in the beginning than it is to be sure everybody puts their curly braces (or whatever) in the same place.

Once you start coding, minimize distractions from that activity by killing petty disagreements (arguments over tab width count as petty, in case you were wondering). This will improve communication (everybody's code will look similar), make it easier to refactor (same reason), and minimize integration headaches due to having people reformat code simply to understand it.

[We need to add the story of "Collective Code Ownership" and "Frequent Pair Changing" caused brackets to change from day to day, until people got tired of it].

None of this is stuff that communication and the other processes can't fix. The de facto coding standard will often emerge if the rest of the things are humming.

[1] *Refactoring*, p. 6. Fowler was referring to design being ugly, but the argument holds for formatting. Formatting isn't as important as design, but bad formatting can slow you down, just like bad design can.

Chapter 23

System Metaphor

asdf.
--

> *A system metaphor is represents a common understanding of the system for everybody involved. It makes design easier, and keeps everyone focused on the goal.*

A picture paints a thousand words. Images, even mental ones, often communicate more than words alone. Consider politics.

Every four years in the United States, we go through this strange ritual called a Presidential Election. The candidates don't waste time on talking about how they will do things once elected. That gets messy and confusing. They try to nail down what they will do in a simple slogan that will capture the attention, and the vote, of key blocks of voters. Assemble enough blocks, win the election. What is that slogan? It is a shorthand way for voters to understand what the candidate wants them to understand about his candidacy. It's a metaphor. Saying "It's the economy, stupid" sells better than explaining macroeconomics.

Why Have A Metaphor At All?

One of the biggest challenges of developing software is having customers and developers talk to one another in a language they both can understand.

XP says that customers should decide the features of the system based on business value, since that's their language. XP also says that programmers should decide how those features get built, since that's their language. What happens when a customer and programmer need to talk, as XP requires them to do – a lot? A metaphor can ease the translation point. A poster on Ward's Wiki put it well:

> *Isn't the intent of the System Metaphor to improve communication among the entire team (customers and non-programmers included) by creating a common way for all to view the system, rather than just expressing an architecture to the programmers?*

> *"The system is a bakery" jives better than "The system interprets text as commands and executes them against builders that produce resultant objects and attach assorted decorators, sorting them via pipes and filters to the correct bins; the user can than browse and eat them as needed" -- RodneyRyan*

Programmers might understand the latter description, but communication with the customer will break down if it's all you've got. It focuses too much on implementation details. A metaphor lets the team focus on "what" rather than "how" in the beginning. If you can find a good metaphor, it can be a powerful tool.

How To Start Creating Metaphors

You can't always find a killer metaphor. Sometimes your metaphor will be a description of what's in your domain, such as a contract management system with contracts and customers. This is what Kent Beck called a "naïve" metaphor. There is nothing wrong with this if it is a shared story for what the system will do that everybody can understand.

A few rules of thumb might be helpful here:

1. Pick a metaphor that's consistent with your domain. This was what the Chrysler project did. Chrysler makes cars, so the project team used a "production line" metaphor with "parts" and "bins". A metaphor about "cakes" and "ovens" wouldn't have made any sense.
2. Don't waste time on finding the perfect metaphor. Find a useful one and go with it. The beauty of a metaphor is not critical. Having everybody understand how the system fits together is critical.
3. Remember the limits of a metaphor. Don't let it hold up your progress. Remember that you can torture a metaphor. If something doesn't quite fit into the metaphor you chose, that's fine.

Why Metaphor Isn't Essential

You have to be developing a system for a metaphor to matter. Once you are in the process of creating that system, a metaphor can be a great tool.

A metaphor can facilitate communication, because everyone is speaking the same language at some level.

A metaphor also facilitates planning and estimating, since common language about what the system "looks like" can reflect common understanding of the sizes of various components, the degree of complexity in having them talk, and so on.

A useful metaphor can help to keep your design simple, since everybody knows where new things fit into the picture.

Chapter 24

40-Hour Week

asdf.
--

> *If you burn the candle at both ends for too long, pretty soon you run out of wax.*

The exact number of hours each member of the team works during any week isn't important. Forty hours is an arbitrary number used to represent a "normal" week where there is a healthy balance between work and non-work. That balance has been forsaken for lots of reasons. Very few of them are any good.

Many people in the IT world have a problem with working too much. With the explosion of eCommerce and the intoxication people seem to have with doing things at "Internet speed", the problem has gotten much worse in recent years.

Working too much is a recipe for failure and disillusionment. It also doesn't produce the desired results.

Working too much usually keeps you from accomplishing the short-term goals you were shooting for in the first place.

Working too much will burn you out in the long run, even if you love your job. This will keep you from accomplishing your goals in the future.

Why People Work Too Much

People work too much when

1. they are hiding from something else in their lives that they can't or don't want to deal with
2. it is a status symbol for them
3. it will help them get ahead, or they think it will
4. they have no choice, usually because somebody with power over them requires them to work too much to meet a deadline

The point is simple. When people work too much, there usually is a deeper problem that hasn't been addressed.

What's Wrong With Burning the Oil?

If you are working too much, or you are forcing others to do so, stop it. Attack the underlying problem and fix it. Not doing that just delays pain.

If there is a problem in some other area of your life that you're running from, take the time to fix it. If you don't, you will regret it. Nobody that we know of has wished on his deathbed that he had spent more time at the office. Most people regret not focusing on the non-work things that have lasting significance once their working lives are done.

If you are in charge of an organization where working too much is some sort of badge of honor, fix that problem. If you work for such an organization, leave. Roy used to think he could work anyone under the table. In the end, he was the one on the floor. Now he thinks that working too much is a sign of insanity, not honor. There is nothing wrong with working hard. There is a problem with killing yourself.

If you are in charge of an organization where working too much lets people get ahead, change the incentive structure. If you work for such an organization, leave. The irony of this "work to get ahead" mentality is that it usually makes you fall behind. Attrition rates are high at Doritos organizations ("Work all you want – we'll make more"), which leads to higher recruiting and training costs, and disrupts client service. High pay can compensate for no life only so long. People are not machines. Working too much will cause other areas of their lives to suffer, which will reduce their effectiveness when they are at work. Even if you love your job, too much of a good thing is unhealthy. Think of a job you love like dessert. It's all right to eat it regularly, but you need to eat other things in order to stay healthy. And you need time for exercise.

"Balance" should be more than a platitude HR groups use to convince prospective employees that they won't die young. It is the only thing that can keep you producing over time. Steven Covey talked about this idea in *Seven Habits of Highly Effective People*. He described the balance between "production" (P) and "production capacity" (PC). Non-work time is when you can focus on the PC side of the equation.

If you burn the oil long enough, sooner or later you run out of oil. Burnout can happen when a person

1. is forced to work more hours than he wants to for an extended period of time
2. works more hours than he should for an extended period of time.

Working more hours is almost never the solution to a productivity problem. It can be a solution to a short-term schedule problem, but it's a very risky one. As the management proverb says, you can't put nine women in a room for a month and make a baby. Tired, overworked people tend to make mistakes and to be unhappy. Their productivity goes down, not up.

The C3 project defined "overtime" as time spent at work when you didn't want to be there. That's as good a definition as we've ever heard. The reality of it will be different for each person, but everyone has his limits.

Sometimes you just have to work more than you want to. Usually that's because you or somebody else screwed up. You underestimated a task or you promised too much. Your boss promised too much, or doesn't have the guts to say no. Think of working overtime like using a credit card. It's more convenient than cash, and it lets you defer payment in the short term. But the interest is a killer. If you build up too much debt, you'll go bankrupt. Overtime is the same way. You can do it sometimes to help smooth things out. If you do it too much, you'll go bankrupt.

How To Start Working "Normal" Hours

If your current work culture isn't used to working "normal" hours, this can be a tough thing to adjust to. This is especially true if some people on the team refuse to cut back.

First, define what is "normal" for you and your team. This depends on your own physiology, as well as on your commitments outside work. Not everyone will hit stride at the same number of hours per week. For some, forty hours is too little. For some, it's too much. It's important for everyone to be on roughly the same work schedule. You can't pair if your better half doesn't show up until you've been at work for an hour.

Second, cut back gradually to that normal schedule. Working too much is a bad habit. It's hard to break a bad habit cold-turkey (if you can, do it). Try cutting back first. Once you've identified your normal week, try to split the difference between that and what you're working now.

Third, practice not compromising on the hours, unless you screwed up. If you screw up, admit the mistake and correct it. Don't stop there. Work on correcting the screw-up next time. Practice being honest during planning. That will support a sane work schedule.

Fourth, have some courage. It takes guts to work a reasonable schedule. When someone demands that you work too much, reflect on the situation to determine if you made a mistake. If you didn't, have the courage to say no.

Increasingly, true balance is something you have to fight for. Sometimes it requires making some tough choices that prick your ego a bit. It is worth the effort.

Why A 40-Hour Week Isn't Essential

You can do XP without working normal weeks. It's just harder. If you do this for too long, any approach to software development, XP or otherwise, will fail.

At first glance, a normal work week might seem to be the oddball XP practice. To understand how it fits in, consider what happens when people on the team consistently work huge amounts of overtime. They get tired. They feel tremendous stress because other areas of their lives get crowded out, and some (like ignored families) push back. That translates into a team full of distracted people prone to make stupid mistakes. The likelihood of such people being effective pairs, seeing opportunities for refactoring, or communicating very well is extremely low.

Mess with people's lives, pay the price.

Chapter 25

Other XP Roles

asdf.
--

asdf.

This is coaching and tracking…

How do these fit in?

Do you always need them?

What about a player/coach?

These are all great questions. We've never heard a convincing argument that says you need to have a coach (an experienced coach always helps, but should you hang it up if you can't get one?) or need to have an official tracker.

It has been argued that you shouldn't have a player/coach. The danger here is that it is hard to do both. The reality is that it's harder for some than others.

Ken seems to be able to switch hats pretty well. He does what makes sense, most of the time. There are times when it is more important for him to get some work done and there are times when it makes more sense to make sure the process is working and people are doing OK. Is he always in the perfect role at any moment of the day, day of the week, or week of the month? Of course not.

Although XP seems to work in fantasy land, there are few things that don't.

We constantly find ourselves having to apply XP in reality. The amazing thing about reality is that it is more like a class than an instance. There are certain things that all realities share (e.g. fixed amount of time), and some things that seem to change (e.g. accepted roles).

If there are people who are on your team that just aren't good developers and don't want to be good developers, figure out if there is another role for them.

If there is an essential thing that needs to happen that no one wants to do or is very good at, find another way to get it done.

Here are two stories… one of Ken's early experience as an XP player/coach. The second is a way we've made the role of the tracker virtual oblivious.

[I need to go back and refactor this… its 1.5 years old]

I am currently leading a team of about 8 that is very "junior heavy" on a significant system which is being written in Java. Greater than half of the team is made up of our client's employees and the rest are employees of RoleModel Software. I am only on the project slightly over half-time. My "second in command" is very bright and has some good Java experience, but is still relatively green. Among the client developers are some very sharp people with little programming experience, and some experienced developers with little OO experience. One of RoleModel's developers is a 19-year old apprentice who is very bright and has an amazing amount of experience for his age, but lacks experience on projects of any size. Another, more senior" developer has joined the project mid-stream and is just now beginning to understand the system. So, on any given story/task, we have a pretty high risk that those driving the task have many weaknesses that outweigh their strengths in bringing the task to completion with a high level of efficiency and quality.

Throughout the rest of this paper, I will refer to those whose weaknesses tend to outweigh their strengths (at the moment) as "junior" no matter what their age or previous programming experience is. In the interest of brevity, I'll refrain from making the explicit distinctions about their skillsets in the rest of this paper.

The project manager (from the client) wants progress on the project, but wants their people to be mentored and take responsibility for tasks. XP, at the surface, seems to be the ideal approach to addressing these simultaneous goals. The reality is that there are certain practices of XP that need to be adjusted to make sure that both actually are getting met.

Reality Strikes Again

I've been the technical lead in environments where I was mentoring 3-8 people at once and felt like I did a pretty good job as long as I stayed alert. Now, when I'm XPing, it should be even easier, right? The tests keep people from screwing up anything. Pairs keep individuals focused on the task at hand and provide two heads which are better than one. Doing the simplest thing that will possibly work keeps the naïve from perpetually having that "deer in the headlights" look. Having a tight loop with the customer keeps

people from going too far in a wrong direction. Monitoring our small item estimates should keep things from going too far astray. Continuous integration keeps away those surprise times where everyone was sure a particular feature that has been working is still working. And the list goes on. I've found that trying to train junior, novice, or even experienced developer's in an XP environment offers its own set of challenges that are often more difficult to manage… but I'm learning.

Although we have had tremendous success with XP, and I am as well-respected on the team as any I've ever been a part of, I find myself feeling unable to prevent people from degrading the software we are building at alarming (to me) rates. Somehow, we manage to have software that is constantly getting better, but it seems as though I'm constantly snuffing out brush fires that threaten the entire forest. For example:

- Developers are commonly ending up "in the ditch" on tasks that they seemed to be "driving steadily down the middle of the road" only an hour earlier.
- Estimates are often significantly off as tasks are confidently oversimplified.
- Pairs are thrashing but think they are doing fine.
- A story is under control until a new partner comes in and puts a twist on a task that makes the task owner panic.

At least currently, there are very few tasks that some of the developers can get their arms around to do a competent job, and they flounder as they try to drive. Although everyone has comparable "load factors", it is clear when the most experienced people are not present who is carrying the load.

Here are some things about XP that I think introduces some unique issues in dealing with more junior developers:

- Since there is no code ownership, the clean code typically degrades when new people get their hands on it, which is fairly often… More and better tests help prevent this, but the new functionality and tests written by the newby can easily introduce unnecessary crud into the stuff that already passed the test. For example, they don't realize that they really don't need the 5 attributes they just added to get their task done, and now the existing code has to manage that extra state information.
- Junior people can't go from CRC (or abstraction) to implementation very well. They seem to get sidetracked very often as soon as they hit

a bump. They don't know how to implement the responsibility, and if their partner is not a lot more competent than them, the blind will lead the blind off the road and into the ditch. Often this is difficult to anticipate, because the newby is confident they know how to implement a responsibility (using the same coding patterns in which they implemented the last one. However, they don't recognize that there is a slight twist necessary because of a subtle nuance of the type of object they are dealing with this time. They haven't seen the variant, and guess at the solution. Often their guess is completely wrong, and they end up with more problems and more bad guesses at solutions. By the time they call in the senior person for help, it takes a lot of time to "pull the car out of the ditch", before getting the car rolling on the road again.

- Having tests written by junior developers can cause as many problems as having code written by them. Junior developers don't seem to be as competent at keeping dependencies out of code they write. So, not only is there code less clean, but their tests are, too. Making a simple change in what you think might be an isolated section of code, suddenly breaks a bunch of tests. Often the tests break only because they were factored poorly (The code they test sometimes actually works as one might expect were the tests not present). So, the tests need to be debugged and refactored. This hurts others estimating ability, because it is hard to guess when you are going to get bit by a test bug.
- If the System Metaphor is not clear at the beginning (which ours was not), it is hard to get the benefit of using it. This is a bigger problem with junior developers as they don't have other experiences to fall back on when they are tackling new stories.
- Even though pairing helps remove egos, more mature (older) people have a tough time admitting they need help from younger people… they'd rather drive in the ditch and deny that they are there.

All in all, we've still had great success with XP and have made more progress than many expected, and greater than I would ever expect given the experience of the team were we using any other development approach I've heard espoused without leaving the junior developers in the dark.

How Have We Handled It?

We are constantly making adjustments… the XP way. It's somewhat of a moving target as the problems we have one iteration change in the next one. Some things get better because we learn from the pain. Some things get better as the experienced developers know the system better. As the System Metaphor becomes clearer, a lot gets better. Additionally, we've done the following.

- Pay as careful attention as possible to the pairs. Whenever possible get a more senior person paired up with a junior one. When it's not possible, make sure a senior person is keeping an eye on the pair to the extent possible.
- Discourage junior people from taking harder tasks during iteration planning, even though they want to… this is often walking on eggshells.
- Try to have a senior person involved at the beginning of every task, even if just for 10 to 20 minutes. Then, have them check back every few hours.
- Encourage the juniors to ask for help outside their partner more often than they would otherwise.
- Use our custom Wiki to be a bit more explicit about the tasks that need to be accomplished. This has been a plus and a minus… not sure what the net is yet.
- The senior people give some chalk talks of 1-2 hours to discuss the design of certain parts of the system… which we've begun to videotape. We do no more than one of these a week.

AutoTracker

[I need to go back and refactor this too… it was written for OOPSLA 2000]

It's impossible to find an objective tracker, so we should not assign anyone the role!

The role of the Tracker was defined early in the life of XP. "The Tracker goes around a time or two a week, asks each Programmer how she's doing, listens to the answer, takes action if things seem to be going off track. Actions include suggesting a CRC session, setting up a meeting with Customer, asking Coach or another Programmer to help." (from http://c2.com/cgi/wiki?ExtremeRoles). In "Extreme Programming

Explained", Kent warns us a Tracker needs the ability to collect information "without disturbing the whole process more than necessary". And, "you can't be such a pain in the neck that people avoid answering you".
At RoleModel Software, we've found that the job of the tracker is typically not one that is desired by anyone we have around. In fact, the kind of person attracted to this job is often exactly the person you don't want… the controlling type who finds fault with anyone who misses a detail and can't see the forest for the trees. My initial reaction was that I'd rather not have a tracker than have the wrong person as a Tracker. On the other hand, not having a Tracker is not really an option. You can't get better at estimating if you don't have feedback on how well you've done on your estimates to date. You can't rely on people to admit they are having a problem with their task because of their natural inclination to hide bad news. Progress is shown by marking off tasks and stories as "completed". Without a chart, how do you know you are making or not making progress?

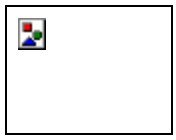
First Attempts at Tracker Automation

When we began using XP, we'd right our estimates on cards, and move them from the "not done" to the "done" board. We could see progress, but if no one asked "how well did you do on your estimates?" as the cards were moved, we were not learning to make better estimates and had no idea how well we were really tracking. Of course, we'd often forget to ask this question. When we did ask the question, we didn't do anything with the data except file it in our heads.
The best way to avoid the wrong people disturbing the process is to automate as much as possible! This seems relatively simple with respect to functional tests. (If we are writing the functional test framework, which we are, we can count how many ran/passed/failed and produce a chart). But what about tracking who is assigned to what and how well they're doing on estimates? We recognized that there was certain data we felt we had to enter (instead of relying on some "big brother time measuring tool")… namely who took responsibility for a task, their estimates, and actual time. As we examined both the people we had and the kind of company we wanted to become (a custom software development company) we recognized that tools would be needed by both ourselves and our clients. We would want the data and statistics to be up to date in pseudo-real-time and accessible from multiple sites. We basically wanted something even more flexible than cards (cards

can only be at one place at a time), but the automation of a computer program (like a spreadsheet) that could roll up the estimates vs. the actual and examine the trends. Then, the coach (as well as the developers and customers) could simply look at the results and make appropriate adjustments.
The answer seemed to indicate a web-based set of tools where data could be entered from multiple sites and viewed from multiple sites with a reasonable level of security. This data would have to be very simple to update and easily associate with stories/tasks. We also didn't want to sacrifice the "token" quality of the cards which identified task ownership.
We thought that a WikiWikiWeb gave us a good way to enter and save stories, break them into tasks, and add data to the tasks as we had it. However, the wiki was text-based, and we feared there was no room for analysis of data. We discussed the possibility of a Wiki-based project management tool with Ward Cunningham (of Cunningham & Cunningham (http://c2.com). Ward agreed to experiment with some way to tally up the results of special fields and add security to a private wiki. In short order the first pass at RoleModel's Online Extreme Project Tracking was born. After several months, another iteration was done and we have been using the version as it stands for more than a year.
Cunningham & Cunningham provides a service (for a monthly fee) to host the special WikiWikiWeb with these features. We create a couple of templates for Stories and Tasks and a special perl script rolls up all the data from numeric fields. *The script looks for any field with numbers in it. Field is defined as a leading space followed by text followed by a colon. This allows us to define new fields as we desire. The reality is that we've pretty much stuck to 3 fields of interest. Since the script does not discriminate, it will also roll up our "as of: 991229" date fields. We simply ignore this side-effect.*

What does it look like?

We use our wiki for all sorts of things about the project. We follow the rule that we create no unnecessary documentation… Most of the necessary documentation is on the wiki, or at least pointed to by it. We can add stories and tasks as necessary and then we organize them by creating a page for each iteration.
For a given iteration, we have a page describing each of the stories/tasks we're tackling. E.g.



http://www.rolemodelsoft.com/
http://www.rolemodelsoft.com/

Iteration Six

Edit

For up to date tracking status, try http:rollup.cgi
The goal of this iteration is...
The following tasks/stories are part of this iteration:

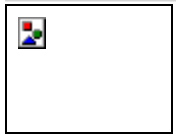
- AddGraphReadingsToNavigatorStory
- AdjustIconsTask
- AddQueriesForUserDefinedFieldsStory

...

Edit

Last edited September 12, 2000
Return to WelcomeVisitors

Then for each story/task, we can break it further into tasks (or not). If we break it into tasks, it looks basically the same as the above... it's just another node in the tree. E.g.



http://www.rolemodelsoft.com/
http://www.rolemodelsoft.com/

Add Graph Readings To Navigator Story

Edit

`http:rollup.cgi`
The goal of this story is to show the readings in graph form whenever something holding a collection of readings is selected in the navigator window.
The following tasks/stories are part of this story:

- AddGraphReadingsNodeTask
- AddGraphReadingsToExistingPerspectivesTask

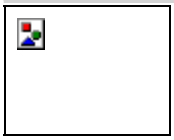
- AddGraphReadingsPerspectiveTask

Edit

Last edited September 12, 2000

Return to WelcomeVisitors

Eventually, we get to the leaves. They use a template (keyed on the suffix 'Task' in our case) when created, and we fill in the details. E.g.



http://www.rolemodelsoft.com/ http://www.rolemodelsoft.com/

Add Graph Readings Node Task

Edit

```
Risk: Low
EstimatedTime: 1.5
ActualTime: 0.75
RemainingTime: 0.5
Developer: KenAuer
Pairs: DuffOmelia, JeffCanna
```

Currently, nodes in the tree stop at Results. We'd like to add a node under t Results node to include Readings. When the Readings are selected, show a which displays those readings... it should be the same graph we currently se when we press the "Graph" button when a Result is selected

Edit

Last edited September 12, 2000

Return to WelcomeVisitors

At the beginning of each iteration, developers who have signed up for tasks enter the EstimatedTime and the RemainingTime (which should start out to be the same).

During the iteration, at the end of each day (or more often), developers update the ActualTime and RemainingTime fields for all of the tasks they worked on... NOTE: ActualTime + RemainingTime will not necessarily equal EstimatedTime.

At any point, if someone wants to see how a story or an iteration is going (with respect to estimates), they can just go to the corresponding "http:rollup.cgi" to see a table which shows the items and the totals. It looks something like this:

---

http://www.rolemodelsoft.com/ http://www.rolemodelsoft.com/

Rollup of IterationSix

	ActualTime	EstimatedTime	Remainin
. . Add Graph Readings Node Task	0.75	1.5	0.5
. . Add Graph Readings To Existing Perspectives Task	0.75	1.5	0.5
. . Add Graph Readings Perspective Task	0	1	1
. Add Graph Readings To Navigator Story total	1.5	4	2
. Adjust Icons Task	0.5	0.25	0
. . Add Queries For User Defined Date Fields Task	2	1.5	0
. . Add Queries For User Defined String Fields Task	1	1	0
. . Add Queries For User Defined Number Fields Task	0	1.5	1.5
. Add Queries For User Defined Fields Story total	3	4	1.5
Iteration Six total	5	8.25	3.5

10 pages examined starting with IterationSix.
Return to WelcomeVisitors

So, I can see at a glance stuff like:

- how much more time is remaining to finish the assigned tasks of the iteration,
- whether we are currently in the ballpark of our estimates,
- which particular story/task(s) are threatening the completion of the iteration,
- whether there's more time than work remaining (or vice-versa).

This ever-present feedback which helps me (as the coach) or anyone else on the team (e.g. the official Tracker) ask some good questions at the next stand-up, such as:

- Has everyone been updating the wiki at the end of the day? (hopefully the answer is "no" when the data that prompted me to ask showed we were behind schedule).
- It looks like the XYZ story has been giving us some problems... is it under control? is the customer aware of the issues?
- It looks like Joe has more tasks than he can finish in the remaining 5 days of the iteration. Joe, how can we help you out?... Mary, it looks like your tasks are done, can you help?

This allows us to make the role of the Tracker a lot less time consuming. It is also much less obtrusive.

What We Like

During development, the unobtrusiveness of the wiki is excellent. During the day, as we understand what we have to do, we can simply push the edit button and add a note or two for ourselves (or the next person to look). At the end of a task or at the end of the day, we simply edit two numbers "Actual Time" and "Remaining Time". We don't need to get grilled by anybody. Before the end of the next day's stand-up meeting, several people usually "roll up" the iteration, and might drill down a bit if something seems to be out of whack. The stand up meeting is usually the place where concerns are communicated.

We also like the idea that the customer and others interested in the progress can look at how things are progressing from wherever they are sitting. Because of the security features (simple login name and password), we can limit exposure as much or as little as we want. The message to everyone around is that we have nothing to hide. We want open communication. Most others don't spend too much time looking at it, but like the idea that they can.

Backups are done every night, so we don't have to worry about where the cards are.
Customers can edit the stories. (In reality we have a problem getting them to do it, but we don't think wiki is the issue here). If we need some sample data, we can call them up and they can update it.
The wiki is also a great place for storing other stuff about the project. When new people come on board, they know that they can go to the wiki to find things they don't feel they can ask or just to study something deeper. (e.g. we have our integration process described there. Even though we walk them through it, some people just feel more secure knowing they can read about it to reinforce what they just saw).

What We Don't Like

Moving the stories around on the wiki is not as easy as moving the cards around. With a card, a story on the back burner is just on the other side of the table and easy to find… you don't even have to remember its name. On the wiki, there has been several times during a discussion when someone has said "don't we have a story for that?"… finding it on the wiki is not easy, especially if you don't remember what you called it. This could theoretically be faster if we organized it better. Psychologically, we've found sorting through cards at the planning game easier than sorting through wiki after the planning game.
We use cards at the planning game, and then someone enters them into wiki afterwards. This gives us a higher "shuffling ability" during the planning game, and let's us track more easily during other times. However, wiki tends to be the "database of record" and the cards don't show up at the iteration planning meetings. We've tried printing out the stuff on the wiki, but 8.5x11 sheets of paper are a poor substitute for smaller index cards. So, we tend to do our iteration planning meetings by just writing candidates for the iteration (based on past planning & performance) on the whiteboard and sorting them there… it just doesn't feel right when you are trying to make trade-offs this way since you can't sort as efficiently when you have to erase and re-write.
We currently don't have a good way to throw out anomalies (other than removing them from the record) or creating pretty charts… on the other hand

Other Thoughts

We have thought of a lot of things we can do with the data we have collected… it just doesn't seem productive.

For example, we could also organize pages by developer (which tasks are assigned to whom), or any other means. We can "rollup" the entire project, or any part of the project by simply creating a new wiki page with bullet items which point to other pages that contain stories/tasks. It doesn't seem to us that any of these organizations are really that valuable and worth the effort (even if the wiki was easier to use than it currently is). For example, if we rolled up individual peoples estimates, the concern would be that there would be incentive to only record numbers that would show yourself to be a good estimator. By not doing it, we instead focus on the tasks that were difficult to estimate, not the person as a poor estimator. We discuss what through off the estimates and, as a team, look out for that next time we estimate… it could have happened to anyone. So instead of picking on the poor chap it happened to, we all learn from the mistakes.

This kind of tracking also helps the focus on how well the team is doing completing an iteration. Although you can drill down on a problem task and find out who has taken responsibility for it, this is typically used to find out who needs help, not to point fingers. There is still the occasional ribbing, but I think the finger would unnecesssarily be pointed at the individual far too often if we tracked that.

Theoretically, we can get a lot of historical metrics also. We can, but we typically haven't. We would if we found much value in it. So far, we have found that rolling up the current iteration tends to give us all the feedback we need to make forward thinking decisions. Some project archaeologist could possibly find some interesting things by analyzing old iterations, but I don't know how valuable they'd be for what purposes. I have lots of guesses, but for a variety of reasons (mostly because the iteration pages tend to get munged at the end of an iteration and we have tended not to think about preserving them before we munge them), I wouldn't have a lot of confidence in much of the numbers.

Conclusion

Story cards and task cards are a great way to plan and operate an XP project. However, they leave a lot to be desired when it comes to tracking. The cost of

a manual tracker seems high considering the return. The AutoTracker we use seems to be the simplest thing that could possibly work and had a lot of side benefits (namely being the developer's source of documentation whenever verbal communication wasn't enough).
Could it improve? Certainly. The main things lacking are

- some number crunching which could automatically produce some charts to show us trends,
- a better way to navigate between tasks (a nicer tree structure).

However, we think the value add for these (and possibly other) useful features is significantly less than we've already experienced.

Section Four: Uncharted Territory

XP is an evolving reality. As it becomes more well-known and more popular, companies not known for being pure innovators are seeing how it can help them leapfrog their competition. That means more people accustomed to "heavy" approaches are beginning to talk about "lightweight" ones.

When folks new to XP start talking about it, especially those with a business focus, they ask questions that often make XP converts uneasy. These are the hard questions. They expose the parts of the discipline that haven't been explored fully yet. They force people to think about XP at the boundaries.

Story about Lewis and Clark would be good…

Chapter 26

Selling XP

asdf.
--

> *Don't sell XP. Sell the results of XP. Then prove that it can do what you said it could, without unacceptable risk to your customer.*

We believe that lighter approaches are the future of software development. XP is the most intuitive and practical lightweight approach we've ever seen. Most of the programmers who try XP love it. Anecdotal evidence suggests that it helps teams produce great software quickly, and that that software is valuable to customers. The problem is that people aren't buying.

The main reason customers resist buying XP projects is fear. The only way to sell XP is overcome that fear. The only way to overcome it is to

- focus on the results of XP (the value that it adds to the customer)
- convince customers to let you prove it
- take their risk away while you prove it

If XP can't live up to its promises, you won't be able to sell it no matter how good a story you tell. If your story is good, though, and you take the customer's risk away, you'll get the opportunity to prove that XP can do what it claims.

Once you get the chance, prove it. Do it a lot. Develop a track record of proof. Then sell that track record applied to each new customer's particular situation.

Why People Don't Buy It

The primary reason customers don't buy projects that use XP is that they are afraid of it. They may give other reasons, but fear (or "risk aversion", if you want to be more polite) underlies them all.

The name "eXtreme Programming" conjures up images of Mountain Dew-drinking snowboarders slinging code with reckless abandon. Customers don't understand the discipline that XP requires. We suffer because of their assumption.

But the name is just an excuse. The root of the problem is that customers are afraid to try something new and different.

Anything that challenges conventional thought within a discipline will seem radical. XP is no exception. The XP practices (especially Pair Programming and Collective Code Ownership) are uncommon in the industry. Even programmers resist them sometimes. Given that, customers have another convenient excuse not to try XP: if they try it, they'll run off their best people.

Traditional approaches have been tried for over thirty years. Teams have used each of them to produce great software. But the overwhelming evidence suggests that teams using heavier approaches don't produce results reliably, they bust the budget, and they deliver late, if they deliver at all. Those approaches simply put too much in the way of a project to be successful. Potential customers who claim that other approaches are "proven" are ignoring these facts. But as we said in the Introduction to this book, these old ways are comfortable because they are acceptable.

How To Sell XP

The primary reason customers are afraid of XP is that the proponents of XP (including us, until recently) haven't focused on selling what customers really care about. Customers care about results in terms of positive impact to their business. That is what we need to be selling.

The only way to sell XP is overcome customers' fear of giving it a try. The only way to overcome that fear is to

- focus on the results of XP (the value that it adds to the customer)
- convince customers to let you prove it
- take their risk away while you prove it
- develop a track record of proof you can show to other potential customers

The Results

Customers don't speak the language of XP. They shouldn't have to. XP should speak the language of business. This means that you shouldn't "sell XP" at all. You should sell the *results* of XP.

Your odds of failure are astronomically high if you try to sell XP itself. No software development organization that we have heard of says that their process stinks, or that they deliver low-quality software, or that they're slow. If you try to

sell XP as a better mousetrap, you will be one more voice in an already crowded field.

Identify your market's needs for value. Sometimes you can quantify that value in terms of dollars, sometimes you can't, but you had better know what it is. Once you've identified it, determine how XP can help you partner with customers to deliver that value. Translate XP into a means of delivering that value. That's what you sell.

What kind of results to potential buyers care about?

- Improved profits
- Improved process
- Repeatable ability to deliver
- Competitive advantage

Of that bunch, competitive advantage is most important. Sustainable competitive advantage is the lifeblood of business. XP can help you give it to your customers. XP supports radical innovation in their products and services by being fast and flexible, and focused on value. They can be faster to market with better products and services. They can respond to customer needs more quickly. They can expand their markets faster and easier. They can reduce the cost of maintaining their software assets over time. That is the compelling financial picture to sell.

If XP can't live up to its promises, you won't be able to sell it no matter how good a story you tell. If your story is good, though, and you take the customer's risk away, you'll get the opportunity to prove it can deliver spectacular results that will make a difference to your customers' businesses.

Proving It

The only way customers will let you prove yourself to them is if you do three things:

1. give them something of value to get you in the door
2. minimize their risk while you are proving it. Once you prove yourself, renegotiate based on that proof.

We accomplish the first goal by charging our customers for an initial Planning Game, which we call The Planning Workshop™. XP says that you shouldn't try to set scope in stone before the project starts, since requirements evolve throughout the project as the team learns. However, customers typically want an approximate cost

and delivery date for the project so they determine if they want to sign up for it. That requires a broad understanding of scope.

Our workshop is a preliminary sizing exercise, much like Beck and Fowler describe in *Planning Extreme Programming*.[1] They recommend that you run a Planning Game at a coarse resolution to help your customer answer the question "Should we invest more?" Simplify the exercise by assuming stories are independent of each other, and that you will develop necessary infrastructure along with each one. Move fast. Guess about some estimates, and leave plenty of padding. What you end up with is what they call "the big plan".

The deliverables for the workshop is just such a big plan. Customers get a set of story cards for the next release, the prioritization of those stories by business value and technical risk, and a first cut at a project plan. The plan contains a ballpark estimate of effort, time, and the number of developers necessary to get the job done. An actionable plan like this, even though it's at a course resolution, is very valuable to customers. As Beck and Fowler suggest,

> *Because they can demonstrate their progress regularly, and because they can react as the market changes, investors and customers believe the company will continue to do great things.*[2]

If the customer wants to start a project, we can sign a contract immediately. Sometimes, though, a customer is skeptical. In those cases, we give our customers an "Exploration Period". This is the first one or two iterations of the project at the rates we would charge the customer if they had already signed a long-term contract with us. This lets us show them that it's worthwhile to have a relationship with us. Once the Exploration Period is over, we deliver the software and tests we produced and ask the customer whether or not he wants to continue.

We accomplish the goal of removing risk by structuring our contracts to focus delivering value from the project and minimizing risk for the customer. Out contracts do this in four ways:

1. We apply have of the customer's cost for The Planning Workshop toward their first iteration after the Exploration Period.
2. We keep the contract period short (2 iterations).
3. We allow the project to change direction as the team members (including the customer representative) learn things.

---

[1] Beck, K. and Fowler, M. *Planning Extreme Programming*, Addison-Wesley, 2000, pp. 35-38.
[2] Beck, K. and Fowler, M. *Planning Extreme Programming*, Addison-Wesley, 2000, p. 38.

4. We give the customer the option to cancel every time the contract is up, if they give us notice of 2 iterations or 30 days, whichever is shorter.
5. We bill by the iteration, so that customers can be sure they are getting production-ready value for their money.

Our contracts look like the ones Kent Beck and Dave Cleal described in their paper *Optional Scope Contracts*.[3] Each one contains a clause like this:

> *The customer will pay the our team of N developers $N/iteration for the next 2 iterations. We will bill the customer by the iteration. The customer may terminate the project at any time by giving us notice equal to 2 iterations or 30 days, whichever is shorter. This contract will renew automatically upon termination, unless explicitly cancelled by the customer.*
>
> *Whatever software is delivered will meet the quality standards defined in Appendix A. The stories, business/technical prioritization of those stories, and an initial project plan are included in Appendix B. All of Appendix B is subject to change during the project as the team learns.*

This is the simplest contract that could possibly work. It aligns the customer's interests with ours. Everybody involved with the project wants to deliver value as soon as possible, and everybody wants the project to continue.

Our customers have a reasonable idea of what they may be getting when the project starts, based on the output of The Planning Workshop™. But what they want, and therefore what they should get, always ends up being different from what they imagined at the beginning. Our contracts force customers to give up the illusion of control over scope at the beginning of our contracts in exchange for getting the software they want at the *end* of a project. And every bit of software they get will be 100% done, as confirmed by automated tests, of course.

Developing A Track Record

Develop a track record of proof. Then sell that track record applied to each new customer's particular situation.

Scrap and claw to get your first customers that will let you do XP projects and build a track record. Track the value that you add. The more you can quantify the

[3] Beck, K. and Clear, D. *Optional Scope Contracts*, Kent Beck and Dave Cleal, 1999.

value that you add, the more you'll be able to attach a price to it that will preserve your margins. Quantify it in terms of three things[4]:

- How much value will you add?
- How soon will you add it?
- How sure can the customer be that you will add it?

If you can't quantify the value that you add in terms of dollars, at least get testimonials describing the value that you added and how happy your customer was with your service.

Once you have a track record, sell from it. Talk to your customer. Know his industry. Know how his critical success businesses and functions contribute to his revenues and costs. Know how you can improve each of those. Find out how far he is from the new levels you can give him, which represent a quantifiable competitive advantage. Then sell him your ability to do that, setting your price at a level that represents an ROI for him that is greater than his hurdle rate (typically 10%-15%) and an attractive cut for you.

If you position yourself as a value-adding partner, you can protect yourself against your competition. If you position yourself as an adder of cost, you will trade away your margins in order to undercut your competitors' price. You have to choose which strategy is better.

Easier Said Than Done

All this talk about selling value is great, but is it that simple? Not really.

A smooth marketing message and a compelling business case help, but they aren't enough. This is because businesses are run by people. People aren't predictable, they don't always do the smart thing, and life isn't always black and white.

The only way you will be successful in selling XP is to develop relationships with your customers. Treat them as your partners. They will see you that way in return, as along as you deliver superior value to them. If you are viewed as a partner, it will be difficult for competitors to unseat you based on price, and nigh on impossible for them to hijack your relationship.

---

[4] The way to quantify value, and the process of selling from "norms" are not original with us. Mack Hanan talks about both in his book *Consultative Selling: The Hanan Formula for High-Margin Sales at High Levels*. It's a great book, although it doesn't address the challenge of quantifying the value added by services businesses very well.

The bottom line is simple. If you partner with your customers to give them the competitive advantage they want and need, they will give *you* the competitive advantage *you* want and need.

Chapter 27

Scaling XP

asdf.
--

> *Don't say XP will not scale until you've tried it. There are creative ways of scaling XP that have not been tried yet.*

"XP doesn't scale." We've heard that many times. Invariably, the person saying it hasn't tried to do it. Sometimes that person hasn't even tried XP. At best, that person is saying, "I don't think XP will scale." If you haven't tried it, your criticism is theoretical. Until you get empirical evidence that XP won't scale, don't throw stones.

Should You Need To Scale?

If you have a project with more than ten or twelve developers, don't jump to the conclusion that XP isn't workable for that project. Instead, ask yourself why you have so many developers. Do you really need them, or are you throwing bodies at the problem? Would ten great developers be enough? What if you had ten good developers, and you got out of their way?

On many projects, more people isn't the answer. In fact, Frederick Brooks' work shows that adding people to a project actually jeopardizes its success.[1]

When To Scale

We can imagine cases where you would need more than ten or twelve developers. Sometimes, you *can* get more done sooner if you have more people working on it. When is that true?

The most obvious case is when it is clear that more could be done sooner if more people were involved.

Another case is when

[1] Frederick Brooks, The Mythical Man Month. Brooks' Law states, "Adding manpower to a late software project makes it later."

How To Scale

XP does seem to fit better with small teams. It's hard to imagine having a stand up meeting with fifty people. We haven't tried it with that many. In fact, all of our projects have had twelve or fewer developers.

If you determine that more people are necessary to accomplish your project's goals, don't be afraid to ramp up. Keep your methodology as light as possible, but proceed with courage. Follow some XP advice in determining how to do this. What is the simplest thing that could possibly work for a large team?

One of the ideas we've heard bounced around has some promise. If you need a fifty-person project, organize it as a set of ten-person teams. Call these XP Teams. Each team will function using XP "by the book". Each will have its own customer that talks to them directly to "drive" that team's efforts (i.e. participate in that team's Iteration Planning, etc.). Things will get a little different when one XP team needs to talk to another for whatever reason.

Create a Coordination Team, whose role is to

- make sure cross-team communication is happening whenever it needs to
- serve as the point of contact for the customers for each XP team
- do Release Planning

The customer from each XP Team is automatically a member of the Coordination Team. Each XP Team should have a representative on the Coordination Team as well. Each XP Team should rotate members through the coordination role so that everyone gets familiar with it, and nobody gets bored.

The Coordination Team does all Release Planning for the project. The customers in the room have to resolve conflicting customer priorities among themselves, perhaps by playing some version of the Senate Game. This exercise views each customer's priorities like a proposed Bill in the United States Senate. Each customer has to fight for passage of the bill (i.e. the set of priorities) he's sponsoring. If there isn't majority support for those priorities, that customer can negotiate with other customers to get the "votes" he needs to have those priorities included. The point is, no developer can decide how to resolve conflicts between customer priorities. That is a business decision only customers can make.

Once the scope of a release is decided, each XP team is responsible for Iteration Planning for its own work during the next iteration. They do this with their own customer…

Do dependencies become more important here?

Does it make sense for XP Teams to be doing Iteration Planning?

Chapter 28

Measuring XP

asdf.
--

asdf.

Many of the claims about XP are theoretical. They make intuitive sense, but they are based on anecdotal evidence. That isn't a bad thing, but it's not enough for industry acceptance. We need to get some numbers for XP.

Does it increase code quality? Along what dimensions? By how much?

Does it reduce the cost of delivery? By how much?

Does it reduce staff turnover? By how much?

Suggestions for Research

If XP is what it claims to be, it should allow teams to deliver better software faster than other approaches. At this point, however, most of the data is anecdotal.

Chapter 31

Other Stuff

asdf.
--

asdf.

Great place to talk about

- language choice
- embedded/distributed systems
- eCommerce projects
- startups
- legacy integration

Asdf

Chapter 32

Where To Next?

The story goes that when Alexander the Great reached the easternmost end of his conquest, he wept because there were no more worlds left to conquer. We are not in that position. The XP adventure is just beginning.

We have been on the road a while. We've discovered some things that work, and others that don't. We hope you can profit from our experiences as we relate them in this book. The question now is where to next?

Resistance to XP is growing. It is a "disruptive idea", very similar to Mort Christensen's "disruptive technology". Existing companies, some of them very well run, are clinging to tried and true methods of developing software. They have applied more rules, guidelines, process improvements, and formality in the hope of increasing their chances of success. We believe those attempts will collapse under the weight of their own complexity.

The future is "lightweight". In a software world where change is both normal and constant, approaches that handle change the best will be the most successful.

Christensen has told us the fate of those well-managed companies that respond too late to disruptive technologies. It is the same with disruptive ideas. If lightweight methods are the future, as we believe they are, there are two alternatives:

1. deny this, stick with "tried and true" methods, and be leapfrogged by your competitors, or
2. get in the game, explore the boundaries, and shape the future

Shape the future, or have it dictated to you. It has always been so. Doing nothing is the same as choosing to let others lead. Risk avoidance is no excuse for not acting. You cannot have change without risk. The future is coming. You can lead, follow, or get out of the way.

The business case for XP has not been made as completely as it needs to be. That is what needs to be done next. The only way to get it done is to experiment at the boundaries of the discipline and to determine what lightweight means in a context broader than projects with a single team of ten or twelve people.

Those companies that participate in helping to make that business case will lead the way to the future of software development. We want to help them explore.

Section Five: Reference

A good reference bookshelf is an invaluable tool. The internet puts mounds of information at your fingertips, but it hides it under a lot of junk. Finding what you need can be a needle-in-a-haystack exercise.

Here is a collection of some reference materials we can't do without. Feel free to add them to your bookshelf.

Laurie/Alistair article

Integration procedures using VA

JAccept™ overview

Filename: xpapplieddraft1.doc
Directory: C:\TEMP
Template: C:\Documents and Settings\xp3\Application Data\Microsoft\Templates\rmsTemplate.dot
Title: “You’re a fool to think this will ever work
Subject:
Author: RoleModel Software Inc.
Keywords:
Comments:
Creation Date: 3/30/01 4:31 PM
Change Number: 2
Last Saved On: 3/30/01 4:31 PM
Last Saved By: RoleModel Software Inc
Total Editing Time: 11 Minutes
Last Printed On: 3/30/01 6:08 PM
As of Last Complete Printing
Number of Pages: 191
Number of Words: 64,663 (approx.)
Number of Characters: 297,453 (approx.)